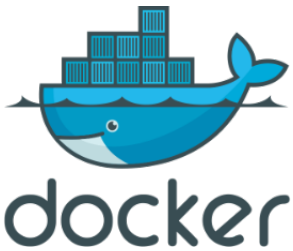
Containers



- Containers Overview
- Accessing Singularity on HPC
- Building a Container
    - Local Builds
    - Remote Builds
- Singularity, Nvidia, and GPU's
    - Pulling Nvidia Images
- Containers Available on HPC
- Sharing Your Containers
- Tutorials
    - Simple Example
    - Running Singularity in a Batch Job
- Example Recipe Files
    - CentOS with Tensorflow
    - MPI

Containers Overview

A container is a packaged unit of software that contains code and all its dependencies including, but not limited to: system tools, libraries, settings, and data. This makes applications and pipelines portable and reproducible, allowing for a consistent environment that can run on multiple platforms.

Shipping containers have frequently been used as an analogy because the container is standard, does not care what is put inside, and will be carried on any ship; or in the case of computing containers, it can run on many different systems.

Docker is widely used by researchers, however, Docker images require root privileges which means they cannot be run in an HPC environment.

Singularity addresses this by completely containing the authority so that all privileges needed at runtime stay inside the container. This makes it ideal for the shared environment of a supercomputer. Even better, a Docker image can be encapsulated inside a Singularity image. Some ideal use cases that can be supported by Singularity on HPC include:

- You already use Docker and want to run your jobs on HPC.
- You want to preserve your environment so a system change will not affect your work.
- You need newer or different libraries than are offered on the system.
- Someone else developed a workflow using a different version of Linux.
- You prefer to use a Linux distribution other than CentOS (e.g. Ubuntu).
- You want a container with a database server like MariaDB.

The documentation here provides instructions on how to either take a Docker image and run it from Singularity, or create an image using Singularity only.

Accessing Singularity on HPC

Singularity is installed on the operating systems of all HPC compute nodes, so can be easily accessed either from an interactive session or batch script without worrying about software modules.

Building a Container

Local Builds

Building a container locally requires root authority which users do not have on HPC. This means you must use a Mac or Linux workstation where you have sudo privileges and Singularity installed. The Sylabs website has instructions that can help users get started on building their own containers. Additionally, Nvidia provides an HPC Container Maker which lets you build a recipe without having to know all the syntax. You will just include the blocks you need (e.g., Cuda or Infiniband) and it will create the recipe that you can use for a build on your local workstation.

Remote Builds

To bypass the issue of needing root privileges to build your container, the Singularity Hub at Sylabs lets you build and keep containers in the cloud as well as share them with other users. You maintain your recipes there and each time you need to pull one, it gets built remotely and is retrieved to your workstation. This conveniently allows you to build containers directly from HPC.

As an example, if you want to build a container in your account, first go to https://cloud.sylabs.io, generate an access token (API key), and save it to your clipboard. Next, log in to an interactive terminal session and find your recipe file. In this example, we'll use the recipe:

```
BootStrap: docker
From: nersc/ubuntu-mpi:14.04

%runscript
    echo "This is what happens when you run the container..."
```

Then, assuming the recipe is stored in our home directory, we can build it remotely using:

```
$ singularity remote login   # paste in your API key at the prompt
$ singularity build --remote ~/nersc.sif ~/nersc.recipe
```

This will produce a .sif file in your home directory that is ready for use.

Singularity, Nvidia, and GPU's

One of the most significant use cases for Singularity is to support machine learning workflows. For information on using GPUs on HPC, see our GPU documentation.

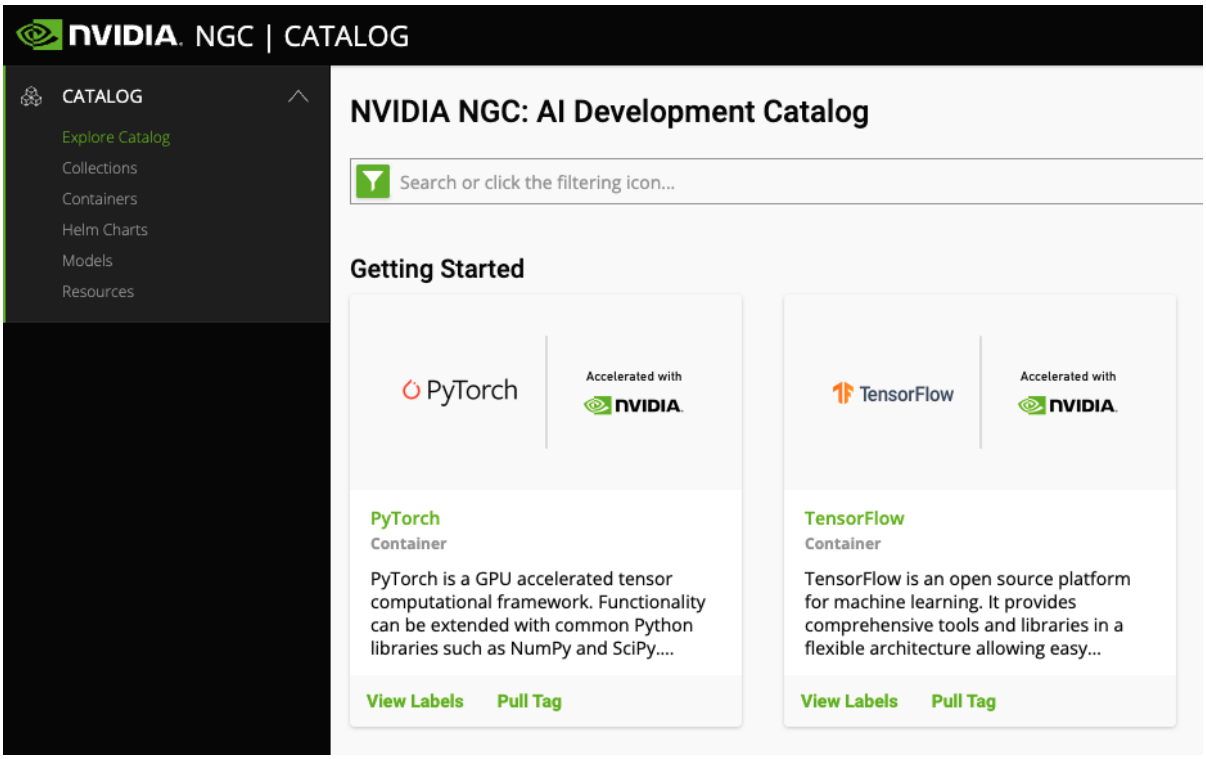
Pulling Nvidia Images

The NVIDIA GPU Cloud (NGC) provides GPU-accelerated HPC and deep learning containers for scientific computing. NVIDIA tests HPC container compatibility with the Singularity runtime through a rigorous QA process. Application-specific information may vary so it is recommended that you follow the container-specific documentation before running with Singularity. If the container documentation does not include Singularity information, then the container has not yet been tested under Singularity.

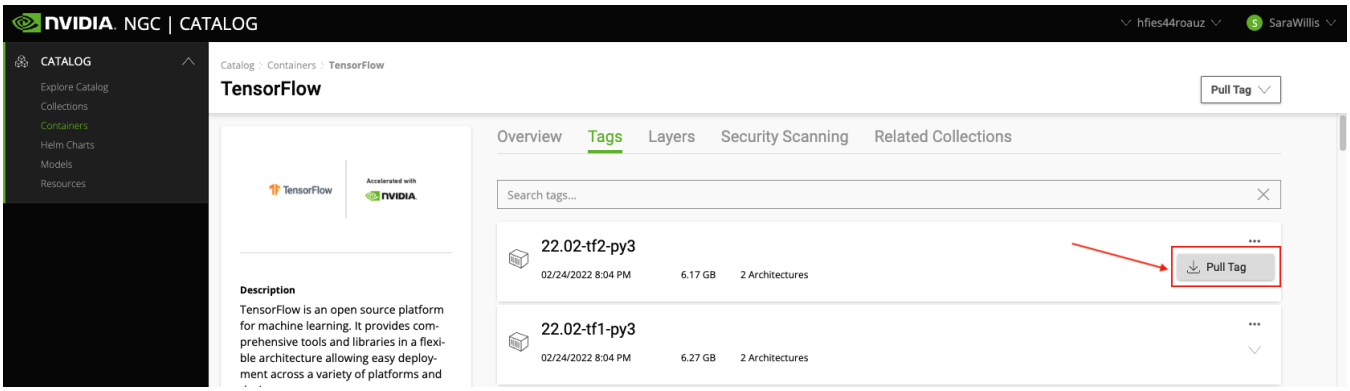
Pulling Images

- The containers from nvidia that are in /contrib have been modified to include path bindings to /xdisk and /groups. They also include the path to the Nvidia commands like `nvidia-smi`.
- Because login nodes are small and do not provide software, singularity images should be pulled and executed on a compute node.

To start, you'll need to register with Nvidia. Once you have an account, you can view their images from their catalogue. Click on the name of the software you're interested in to view available versions



If you click on the **Tags** tab at the top of the screen, you'll find the different versions that are available for download. For example, if we click on TensorFlow, we can get the pull statement for the latest tag of TensorFlow 2 by clicking the ellipses and selecting Pull Tag.



This will copy a `docker pull` statement to your clipboard, in this case:

```
docker pull nvcr.io/nvidia/tensorflow:22.02-tf2-py3
```

To pull and convert this NGC image to a local Singularity image file, we'll convert this to:

```
singularity build ~/tensorflow2-22.02-py3.sif docker://nvcr.io/nvidia/tensorflow:22.02-tf2-py3
```

The general format for any pull you want to do is:

```
$ singularity build <local_image_name> docker://nvcr.io/<registry>/<app:tag>
```

This Singularity build command will download the app:tag NGC Docker image, convert it to Singularity format, and save it to the local filename local_image_name.

Running

Directory access:

Singularity containers are themselves ostensibly read only. In order to provide application input and output host directories are generally bound to the container, this is accomplished through the Singularity -B flag. The format of this flag is `-B <host_src_dir>:<container_dst_dir>`. Once a host directory, `host_src_dir`, is bound into the container you may interact with this directory from within the container, located at `container_dst_dir`, the same as you would outside the container.

You may also make use of the `--pwd <container_dir>` flag, which will be used to set the present working directory of the command to be run within the container.

Ocelote does not support filesystem overlay and as such the container_dst_dir must exist within the image for a bind to be successful. To get around the inability to bind arbitrary directories $HOME and /tmp are mounted in automatically and may be used for application I/O.

GPU support:

All NGC containers are optimized for NVIDIA GPU acceleration so you will always want to add the `--nv` flag to enable NVIDIA GPU support within the container.

Standard run command:

The Singularity command below represents the canonical form that will be used on the Ocelote cluster.

```
$ singularity exec --nv --pwd <work_dir> <image.simg> <cmd>   # <work_dir> should be set to either $HOME or /tmp
```

Containers Available on HPC

We support the use of HPC and ML/DL containers available on NVIDIA GPU Cloud (NGC). Many of the popular HPC applications including NAMD, LAMMPS and GROMACS containers are optimized for performance and available to run in Singularity on Ocelote or Puma. The containers and respective README files can be found in /contrib/singularity/nvidia. But. They are only available from compute nodes, so start an interactive session if you want to view them.

We do not update these very often as it is time consuming and some of them change frequently. So we encourage you to pull your own from Nvidia:

Register with Nvidia at https://ngc.nvidia.com/signin
Obtain an API key
Translate their tag for docker to something like this example:
singularity build tensorflow-20.08-tf2-py3.simg docker://nvcr.io/nvidia/tensorflow:20.08-tf2-py3

- The Nvidia images have been modified to include bindings for your /xdisk and /groups directories if you want to run your jobs there
- The filename has a tag at the end that represents when it was made. For example, 18.01 is January 2018.

Container	Description
nvidia-caffe.20.01-py3.simg	Caffe is a deep learning framework made with expression, speed, and modularity in mind. It was originally developed by the Berkeley Vision and Learning Center (BVLC)
nvidia-gromacs.2018.2.simg	
nvidia-julia.1.2.0.simg	
nvidia-lammps.24Oct2018.sif	

nvidia-namd_2.13-multinode.sif	
nvidia-pytorch.20.01-py3.simg	PyTorch is a Python package that provides two high-level features: • Tensor computation (like numpy) with strong GPU acceleration • Deep Neural Networks built on a tape-based autograd system
nvidia-rapidsai.sif	
nvidia-relion_2.1.b1.simg	
nvidia-tensorflow_2.0.0-py3.sif	TensorFlow is an open source software library for numerical computation using data flow graphs. TensorFlow was originally developed by researchers and engineers working on the Google Brain team within Google's Machine Intelligence research organization for the purposes of conducting machine learning and deep neural networks research.
nvidia-theano.18.08.simg	Theano is a Python library that allows you to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently.

Sharing Your Containers

If you have containers that you would like to share with your research group or broader HPC community, you may do so in the space `/contrib/singularity/shared`. Note that this location is only accessible on a compute node either in an interactive session or batch script.

To do this, start an interactive session and change to `/contrib/singularity/shared`:

```
(elgato) [user@junonia ~]$ interactive -a YOUR_GROUP
Run "interactive -h for help customizing interactive use"
Submitting with /usr/local/bin/salloc --job-name=interactive --mem-per-cpu=4GB --nodes=1    --ntasks=1 --
time=01:00:00 --account=YOUR_GROUP --partition=standard
salloc: Pending job allocation 308349
salloc: job 308349 queued and waiting for resources
salloc: job 308349 has been allocated resources
salloc: Granted job allocation 308349
salloc: Waiting for resource configuration
salloc: Nodes cpu1 are ready for job
[user@cpu1 ~]$ cd /contrib/singularity/shared
```

Next, create a directory, set the group ownership, and set the permissions. For example, if you wanted your directory to only be writable by you and be accessible to the whole HPC community, you could run (changing `user` and `YOUR_GROUP` to match your own desired directory name and HPC group, respectively):

```
[user@cpu1 shared]$ mkdir user
[user@cpu1 shared]$ chgrp YOUR_GROUP user/
[user@cpu1 shared]$ chmod 755 user/
[user@cpu1 shared]$ ls -ld user/
drwxr-sr-x 2 user YOUR_GROUP 0 Apr 11 14:17 user/
```

Next, add any images you'd like to share to your new directory, for example:

```
[user@cpu1 shared]$ cd user/
[user@cpu1 user]$ singularity pull ./hello-world.sif shub://vsoch/hello-world
INFO:    Downloading shub image
59.8MiB / 59.8MiB [==============================] 100 % 4.8 MiB/s 0s
[user@cpu1 user]$ ls
hello-world.sif
```

As soon as your images are in this location, other HPC users can access them interactively or in a batch script. An example batch job is shown below:

singularity_example.slurm

```
#!/bin/bash
#SBATCH --job-name=singularity_contrib_example
#SBATCH --account=YOUR_GROUP
#SBATCH --partition=standard
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --time=00:01:00

singularity run /contrib/singularity/shared/user/hello-world.sif
```

Submitting the job and checking the output:

```
(elgato) [user@junonia ~]$ sbatch singularity_example.slurm
Submitted batch job 308351
(elgato) [user@junonia ~]$ cat slurm-308351.out
RaawwWWWWWRRRR!! Avocado!
```

Tutorials

- The Sylabs GitHub site has files and instructions for creating sample containers.
- Our Github repository has Singularity examples available that can be run on HPC.

Simple Example

The lolcow image is often used as the standard "hello world!" introduction to containers and is described in Singularity's documentation. To follow their example, first start by logging into an interactive terminal session and pull the image:

```
$  singularity pull docker://godlovedc/lolcow
INFO:    Converting OCI blobs to SIF format
INFO:    Starting build...
Getting image source signatures
[...]
Writing manifest to image destination
Storing signatures
INFO:    Creating SIF file...
INFO:    Build complete: /home/uxx/netid/.singularity/cache/oci-tmp
/a692b57abc43035b197b10390ea2c12855d21649f2ea2cc28094d18b93360eeb/lolcow_latest.sif
```

This will pull the image from Docker Hub and save it in your home in a hidden directory `.singularity`. Next, run the image simply using `singularity run`

```
$ singularity run lolcow_latest.sif
 _______________________________________
/ Perilous to all of us are the devices \
| of an art deeper than we ourselves    |
| possess.                              |
|                                       |
| -- Gandalf the Grey [J.R.R. Tolkien,  |
\ "Lord of the Rings"]                  /
 ---------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Running Singularity in a Batch Job

Running a job with Singularity is as easy as running other jobs, simply include your resource requests, and include any commands necessary to execute your workflow. For more detailed information on creating and running jobs, see our SLURM documentation or Puma Quick Start. An example script might look like:

```
#!/bin/bash
#SBATCH --job-name singularity-job
#SBATCH --account=your_pi
#SBATCH --partition=standard
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --time=01:00:00

date
singularity exec --nv dockerTF.img python TFlow_example.py
date
```

Example Recipe Files

CentOS with Tensorflow

centosTflow.def

```
BootStrap: yum
OSVersion: 7
MirrorURL: http://mirror.centos.org/centos-%{OSVERSION}/%{OSVERSION}/os/$basearch/
Include: yum
# best to build up container using kickstart mentality.
# ie, to add more packages to image,
# re-run bootstrap command again.
# bootstrap on existing image will build on top of it, not overwriting it/restarting from scratch
# singularity .def file is like kickstart file
# unix commands can be run, but if there is any error, the bootstrap process ends
%setup
   # commands to be executed on host outside container during bootstrap
%post
     # commands to be executed inside container during bootstrap
     # add python and install some packages
    yum -y install vim wget python3 epel-release
     # install tensorflow
    pip3 install --upgrade pip
    pip3 install tensorflow-gpu==2.0.0-rc1
     # create bind points for storage.
    mkdir /xdisk
    mkdir /groups
    exit 0
# %runscript
   # commands to be executed when the container runs
# %test
   # commands to be executed within container at close of bootstrap process
   python --version
```

To build and test a container from the recipe from an interactive session on a GPU node:

```
$ singularity build centosTflow.sif centosTflow.def # Remember, you will need to either build this on a
workstation where you have root privileges or will need to user a --remote build
$ singularity exec --nv centosTFlow.simg python3 TFlow_example.py
```

As a tensorflow example, you could use the following script:

TFlow_example.py

```
#Linear Regression Example with TensorFlow v2 library

from __future__ import absolute_import, division, print_function
#
import tensorflow as tf
import numpy as np
rng = np.random
#
# Parameters.
learning_rate = 0.01
training_steps = 1000
display_step = 50
#
# Training Data.
X = np.array([3.3,4.4,5.5,6.71,6.93,4.168,9.779,6.182,7.59,2.167,
              7.042,10.791,5.313,7.997,5.654,9.27,3.1])
Y = np.array([1.7,2.76,2.09,3.19,1.694,1.573,3.366,2.596,2.53,1.221,
              2.827,3.465,1.65,2.904,2.42,2.94,1.3])
n_samples = X.shape[0]
#
# Weight and Bias, initialized randomly.
W = tf.Variable(rng.randn(), name="weight")
b = tf.Variable(rng.randn(), name="bias")

# Linear regression (Wx + b).
def linear_regression(x):
    return W * x + b

# Mean square error.
def mean_square(y_pred, y_true):
    return tf.reduce_sum(tf.pow(y_pred-y_true, 2)) / (2 * n_samples)

# Stochastic Gradient Descent Optimizer.
optimizer = tf.optimizers.SGD(learning_rate)
#
# Optimization process.
def run_optimization():
# Wrap computation inside a GradientTape for automatic differentiation.
    with tf.GradientTape() as g:
        pred = linear_regression(X)
        loss = mean_square(pred, Y)

    # Compute gradients.
    gradients = g.gradient(loss, [W, b])

    # Update W and b following gradients.
    optimizer.apply_gradients(zip(gradients, [W, b]))
#
# Run training for the given number of steps.
for step in range(1, training_steps + 1):
    # Run the optimization to update W and b values.
    run_optimization()

    if step % display_step == 0:
        pred = linear_regression(X)
        loss = mean_square(pred, Y)
        print("step: %i, loss: %f, W: %f, b: %f" % (step, loss, W.numpy(), b.numpy()))
```

MPI

Singularity supports MPI pretty well since, by default, the network is the same inside and outside the container. The more complicated bit is making sure that the container has the right set of MPI libraries. MPI is an open specification, but there are several different implementations (OpenMPI, MVAPICH2, and Intel MPI to name three) with some non-overlapping feature sets. If the host and container are running different MPI implementations, or even different versions of the same implementation, hilarity may ensue.

The general rule is that you want the version MPI inside the container to be the same version or newer than the host. You may be thinking that this is not good for the portability of your container and you are right. Containerizing MPI applications is not terribly difficult with Singularity, but it comes at the cost of additional requirements for the host system.

In this example, the infiniband pieces are installed and then the MVAPICH version of MPI. When the job is run, the script will need to load the correct module with the matching version of MVAPICH.

MPI Recipe File

```
BootStrap: debootstrap
OSVersion: xenial
MirrorURL: http://us.archive.ubuntu.com/ubuntu/


%runscript
    echo "This is what happens when you run the container..."


%post
    echo "Hello from inside the container"
    sed -i 's/$/ universe/' /etc/apt/sources.list
    apt update
    apt -y --allow-unauthenticated install vim build-essential wget gfortran bison libibverbs-dev libibmad-
dev libibumad-dev librdmacm-dev libmlx5-dev libmlx4-dev
    wget http://mvapich.cse.ohio-state.edu/download/mvapich/mv2/mvapich2-2.1.tar.gz
    tar xvf mvapich2-2.1.tar.gz
    cd mvapich2-2.1
    ./configure --prefix=/usr/local
    make -j4
    make install
    /usr/local/bin/mpicc examples/hellow.c -o /usr/bin/hellow
```