**Universal Guide for HPC (High Performance Computing) at BinAC or similar Computation-Cluster using Containerization**

**by Maximilian Hilbert[[1]](#footnote-1)**

1. **Introduction**

Due to rights management on the cluster, GPU support using CUDA is difficult to achieve in contrast to a local computer. Furthermore, specialized software dependencies, e. g. tensorflow versions are not useable. A possible solution to cope with this problem is using a containerization method. The **BinAC**[[2]](#footnote-2)Cluster in Tübingen or **desy**[[3]](#footnote-3)in Hamburg support **Singularity[[4]](#footnote-4)** as a containerization software.

The idea behind containerization is, that one can hold the entirety of software dependencies, required at runtime (incl. the operating system), in one file, the so-called *container*. This way one can build the container locally on a system with root access and upload it to the cluster. At runtime all software one needs is *inside* the container, therefore one can also install content that would require root user rights on the local machine.

In the case presented here we consider **centOS**[[5]](#footnote-5) (small operating system), CUDA (Support for Nvidia GPUs), **CuDnn[[6]](#footnote-6)** (Deep Neural Network library), **miniconda**[[7]](#footnote-7) (python incl. software dependencies and updates) and **mlreflect**[[8]](#footnote-8) (incl. Tensorflow and other functions) and finally the code we want to run on the cluster cluster\_job.py, which requires the above-mentioned software dependencies.

1. **Installation**

First, one must make sure to have access to a linux-based machine with root permissions. In this case we use **VMware Workstation**[[9]](#footnote-9) (Virtual machine) with **ubuntu**[[10]](#footnote-10) (linux operating system with graphical interface).

When installed ubuntu on VM (or use a full installation) one can install Singularity via the command prompt[[11]](#footnote-11).

Now create a new file via the build-in text editor, ending with .def containing the so-called definitions of our container. For a detailed explanation on the different sections see[[12]](#footnote-12).

definitions.def

BootStrap: docker

From: nvidia/cuda:10.1-cudnn7-devel-centos7

%runscript

exec "$@"

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

source /conda/etc/profile.d/conda.sh

conda activate test\_env

python --version

python cluster\_job.py

%post

echo "Hello from inside the container"

yum install wget -y

wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86\_64.sh -O miniconda.sh

bash miniconda.sh -b -p conda

source /conda/etc/profile.d/conda.sh

conda update -y -n base conda

conda create -y -n bachelor\_env python=3.6

rm miniconda.sh -f

conda activate test\_env

conda install pip

pip install refl1d

pip install mlreflect

conda install scikit-learn

%test

source /conda/etc/profile.d/conda.sh

conda activate test\_env

python --version

%files

cluster\_job.py

A summary: The definitions include the OS and CUDA, as well as CuDnn in the header of the definition file, as another docker-container. It is **important** to fit the CUDA-Version to the CuDnn and the centOS version and the corresponding Tensorflow/PyTorch version, to achieve compatibility.[[13]](#footnote-13)

*%runscript*

Defines what happens when the commands “*singularity run singularity\_container.sif*” or *“singularity exec singularity\_container.sif* *[command inside container]“* are used on a linux machine to run/execute the container (for explanation on exec, see later). In this case it is important to link to the conda path and activate the environment created in the %post-section, check the python version to be sure, the code is running inside the container.

*%post*

This is the part that gets installed when building the container on our local machine. One can add software dependencies as one needs. Also create a conda environment using the right python version. Keep in mind that centOS comes with almost nothing, so installing wget should be the first line. Be aware of the versions (mlreflect installs Tensorflow 2.1.4, which is compatible with Cuda 10.1, CuDnn 7 and centOS 7 and python 3.6).

*%test*

Is automatically ran after the building process is done to check if a command works. In our case it checks the python version. Keep in mind to first activate the environment.

*%files*

Includes the code one wants to run, although the file can lie outside of the container.

Finally, one can now build the container via the command “*sudo singularity build first\_container.sif definitions.def*”. Making sure one has “*sudo*” (root) rights on the local machine.

1. **Run on cluster**

After the building process, test the container locally, before uploading! First, try to run a *cluster\_job.py* file that only creates a *test.txt* file that pastes the current tensorflow or python version. You should also paste information about your CUDA state there, use “*tf.config.list\_physical\_devices('GPU')*”. If everything works one can upload the container to the cluster, create a *job.sh* file that navigates to the correct path, includes your job specifics (walltime, nodes, etc.) and runs the container via “*singularity run --nv singularity\_container.sif*”. Make sure to use “--nv” which stands for CUDA support. When starting the container and tensorflow one should see the success logs from CUDA. Make also sure to define how many gpus you need in the header of the job.sh file and place the job in the correct queue. The specifics of the header file can change when heading to another cluster that does not use PBS, check the corresponding documentation then for further information.

job.sh

#PBS -l nodes=2:ppn=5:gpus=4

#PBS -l walltime=03:30:00

#PBS -l mem=20gb

#PBS -N batch

#PBS -j oe

#PBS -o LOG

#PBS -q gpu

cd /beegfs/work/**USERNAME**

singularity run --nv singularity\_container.sif

1. **Batch jobs**

If one wants to compute complex and time-consuming jobs, it is necessary to split up the problem in smaller chunks and place it in the queue individually. Due to the fact, that the cluster will always prefer smaller jobs that fit in computation time or capacity gaps. The following example will be dealing with hyperparameter search. Essentially, it is necessary to run cluster\_job.py many times with different parameters. We can hand over a set of parameters to the singularity container, using “*singularity exec --nv singularity\_container.sif ./wrapper.sh param\_1 param\_2*”. This uses the exec-command which pastes a command inside the container and runs it there. In this case it would run a *wrapper.sh* script handing over 2 parameters and starts cluster\_job.sh with these parameters.

wrapper.sh

source /conda/etc/profile.d/conda.sh

conda activate test\_env

python --version

python cluster\_job.py $1 $2

Now one needs to create a bunch of job\_i.sh files that each run wrapper.sh. You can build a simple create\_jobs.sh containing a python script to generate them. All of them should look like the same, where the only difference is the param1, param2 value.

job\_i.sh

#PBS -l nodes=2:ppn=5:gpus=4

#PBS -l walltime=03:30:00

#PBS -l mem=20gb

#PBS -N batch

#PBS -j oe

#PBS -o LOG

#PBS -q gpu

cd /beegfs/work/**USERNAME**

singularity exec --nv singularity\_container.sif ./wrapper.sh 0.001 40

Analogously, on the desy Cluster using slurm as batch-management system

#!/bin/bash

#SBATCH --partition=allgpu

#SBATCH --time=0-03:00:00

#SBATCH --nodes=1

#SBATCH --constraint=GPU

#SBATCH --chdir /beegfs/desy/user/**USERNAME**/slurm\_out

#SBATCH --job-name testi

#SBATCH --output testi-%j.out

#SBATCH --error testi-%j.err

unset LD\_PRELOAD

source /etc/profile.d/modules.sh

cd /beegfs/desy/user/**USERNAME**

singularity exec --nv --bind /beegfs/desy/user/**USERNAME**/beegfs/desy/user/**USERNAME**/singularity\_container.sif /beegfs/desy/user/**USERNAME**/run.sh 0.001 40 35 70 8 4

Here it is important to bind the path of your working directory, so that the container can find the script and singularity can find your container in the first place.

Now you one can install python on one of the login nodes and run a simple chain.py script that runs all job\_i.sh files generated and places the jobs in the correct queue. First, try to do the procedure with one job\_i.sh file to check if the chain works and wait until completed to get an idea of the correct walltime. Always use the smallest walltime and hardware specifics necessary.

chain.py

#!/bin/bash

import subprocess

import os

for filename in os.listdir("jobs/"):

subprocess.run("qsub -q gpu jobs/”+filename, shell=True) [for desy use “sbatch jobs/”+filename]

1. maximilian.hilbert@student.uni-tuebingen.de [↑](#footnote-ref-1)
2. https://wiki.bwhpc.de/e/Category:BwForCluster\_BinAC [↑](#footnote-ref-2)
3. https://confluence.desy.de/display/MXW/Maxwell+Cluster [↑](#footnote-ref-3)
4. https://sylabs.io/guides/3.0/user-guide/quick\_start.html [↑](#footnote-ref-4)
5. https://www.centos.org/ [↑](#footnote-ref-5)
6. https://developer.nvidia.com/cudnn [↑](#footnote-ref-6)
7. https://docs.conda.io/en/latest/miniconda.html [↑](#footnote-ref-7)
8. https://pypi.org/project/mlreflect/ [↑](#footnote-ref-8)
9. https://www.vmware.com/de/products/workstation-player.html [↑](#footnote-ref-9)
10. https://ubuntu.com/tutorials/install-ubuntu-desktop#1-overview [↑](#footnote-ref-10)
11. https://github.com/apptainer/singularity/blob/master/INSTALL.md [↑](#footnote-ref-11)
12. https://sylabs.io/guides/3.0/user-guide/definition\_files.html [↑](#footnote-ref-12)
13. https://www.tensorflow.org/install/source#tested\_build\_configurations [↑](#footnote-ref-13)