Python Workflow

Most of this document uses Patrick Pineault's 'The Good Research Code Handbook' . The sections about using NYU's HPC Greene cluster are from: 'Logging into HPC Clusters within NYU Network on Mac' . Transferring files to HPC Clusters from a Mac/Linux . Slurm Job Scheduler .

What is missing from this document: running JAX on the cluster.

Section A: Making the directory

1. Create the local directory called 'mydirectory'

```
mkdir mydirectory
```

2. Link the directory to Github inside VSCode

3. Create a Python virtual environment using conda

```
conda create -n envname python=3.11.4
conda activate envname
pip install jaxlib
pip install jax
conda list
conda info --envs
```

4. Initialise the environment in VSCode

Inside VSCode, click on 'Python' on the bottom right. Select 'Enter interpreter path'. Use the path from the `conda info --envs` step. Test the Python interpreter using a dummy `file.py` execution.

5. Export the environment; load the environment

Export:

```
conda env export > environment.yml
```

Recover:

```
conda env --name recoveredenv --file environment.yml
```

6. Make the project skeleton

```
mkdir {data,docs,results,scripts,src,tests}
```

- `data`: Where you put raw data for your project. You usually won't sync this to source

control, unless you use very small, text-based datasets (< 10 MBs).
- `docs`: Where you put documentation, including Markdown and reStructuredText (reST).

Calling it docs makes it easy to publish documentation online through Github pages.
- `results`: Where you put results, including checkpoints, hdf5 files, pickle files, as well as

figures and tables. If these files are heavy, you won't put these under source control.
- `scripts`: Where you put scripts - Python and bash alike - as well as .ipynb notebooks.
- `src`: Where you put reusable Python modules for your project. This is the kind of python

code that you import.
- `tests`: Where you put tests for your code. We'll cover testing in a later lesson.

Section B: Saving and loading Python scripts

The project directory should include a few more files

```
mydirectory
  - src
    - __init__.py
    - quantum.py
  - setup.py
```

We can create these files from the command line

1. Initilisation file

```
touch __init__.py
```

2. Setup file

```
touch setup.py
nano setup.py
```

```
from setuptools import find_packages, setup
setup(
  name="src",
  packages=find_packages()
)
```

3. Run the packages

```
pip install -e .
```

4. Test the packages

```
python
import src.quantum
src.quantum
exit()
```

```
from src.quantum import *
print(sigma)
```

Section C: Running programs on NYU's Cluster

In this section, we run a Python file called `file.py` by executing the SBATCH file `run.s`

1. Creating the SBATCH file

```
touch run.s
```

In VSCode:

```
#!/bin/bash
#SBATCH --nodes=1
...
#SBATCH --output=slurm

module purge
module load python/intel/3.11.4
python ./file.py
```

I should set the exact SBATCH parameters based on my needs (GPU, RAM, CPU). The file `file.py` is what I want the cluster to run. I need to purge the node before loading python to ensure that I run the correct version.

2. Copy files to the cluster

```
cd dev/mydirectory
scp run.s file.py vv2102@greene.hpc.nyu.edu:~
... Enter password
ssh-keygen vv2102@greene.hpc.nyu.edu
ssh vv2102@greene.hpc.nyu.edu
... Enter password
mkdir test
mv run.s test/run.s
mv file.py test/file.py
```

To copy an entire folder, we use a similar command

```
scp -r mydirectory vv2102@greene.hpc.nyu.edu:~
```

3. Run the SBATCH file

```
cd test
sbatch run.sP
```

Section D: Running JAX on the Greene Cluster

This section follows NYU's guide, which you can find here.

Running Python packages that are non-native to the Greene cluster, such as `JAX`, requires `Singularity`. In this document, I install `JAX` on the Greene cluster.

1. Create the project's directory in `/scratch/...`

Login to the cluster using the usual steps.

```
% ssh vv2102@greene.hpc.nyu.edu
# enter password
```

Now make the directory, which we call 'jax-example'. Note that my username is 'vv2102'.

```
$ mkdir /scratch/vv2102/jax-example
```

2. Copy an appropriate gzipped overlay image

```
$ cd /scratch/vv2102/jax-example
$ cp -rp /scratch/work/public/overlay-fs-ext3/overlay-15GB-500K.ext3.gz .
$ gunzip overlay-15GB-500K.ext3.gz
```

This copies a 15 GB and 500 000 file capacity overlay 'overlay-15GB-500K.ext3.gz' into the 'jax-example' folder. Overlay images allocate disk space.

We could have used a different overlay, which we can find inside the cluster's directory:

```
$ ls /scratch/work/public/overlay-fs-ext3
```

3. Load a `Singularity` container into the image

We load `CUDA 12` on `Ubtuntu` (`cuda12.1.1-cudnn8.9.0-devel-ubuntu22.04.2.sif`):

```
$ singularity exec --overlay overlay-15GB-500K.ext3:rw /scratch/work/public/
singularity/cuda12.1.1-cudnn8.9.0-devel-ubuntu22.04.2.sif /bin/bash
```

This line runs the `Singularity` application, opens the 15 GB `.ext3` overlay image we created, loads the CUDA 12 `.sif` container into the image, and saves the `.ext3` image.

We could have used a different `Singularity` container, which we can find inside the cluster's directory:

```
$ ls /scratch/work/public/singularity/
```

3. Installing `miniconda` inside `Singularity`

```
Singularity> wget https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-
x86_64.sh
# downloads files
Singularity> bash Miniconda3-latest-Linux-x86_64.sh -b -p /ext3/miniconda3
# downloads files
Singularity> rm Miniconda3-latest-Linux-x86_64.sh
# Remove unnecessary file
```

4. Create the `conda` wrapper

We place the wrapper insider `/ext3/env.sh`. The file does not exist yet, so we must create it.

```
Singularity> touch /ext3/env.sh
Singularity> nano /ext3/env.sh
```

The file contents of `/ext3/env.sh` are

```
#!/bin/bash

source /ext3/miniconda3/etc/profile.d/conda.sh
export PATH=/ext3/miniconda3/bin:$PATH
export PYTHONPATH=/ext3/miniconda3/bin:$PATH
```

Finally, we install all the relevant conda packages:

```
Singularity> source /ext3/env.sh
# activates conda
Singularity> conda update -n base conda -y
Singularity> conda clean --all --yes
Singularity> conda install pip -y
Singularity> conda install ipykernel -y # Note: ipykernel is required to run as a
kernel in the Open OnDemand Jupyter Notebooks
```

We can verify that everything is up to date

```
Singularity> conda  --version
#conda
23.7.2
Singularity> python --version
#Python
3.11.4
Singularity> pip --version
#pip 23.2.1 from /ext3/miniconda3/lib/python3.11/site-packages/pip (python 3.11)
```

Exit `Singularity`.

```
Singularity> exit
```

5. Installing `JAX` (or other packages)

This is the only step not provided in NYU's guide. It is critical that `JAX` and the GPU have the same `CUDA` version. In my example, my GPU is `CUDA 12`, which was indicated in the `Singularity` container we loaded (`cuda12.1.1-cudnn8.9.0-devel-ubuntu22.04.2.sif`), so I shall install the `CUDA 12` version of `JAX`.

Official `JAX` installation instructions are here.

First, I run my image on the cluster:

```
$ srun --cpus-per-task=2 --mem=10GB --time=04:00:00 --pty /bin/bash

# srun: job 36535589 queued and waiting for resources
# wait to be assigned a node
# srun: job 36535589 has been allocated resources

$ singularity exec --overlay overlay-15GB-500K.ext3:rw /scratch/work/public/
singularity/cuda12.1.1-cudnn8.9.0-devel-ubuntu22.04.2.sif /bin/bash

Singularity> source /ext3/env.sh
# activate the environment
```

Second, I install `JAX` (and a few other packages) using `pip` :

```
Singularity> pip install --upgrade pip
# Requirement already satisfied: pip in /ext3/miniconda3/lib/python3.11/site-
packages (23.2.1)

# CUDA 12 installation
# Note: wheels only available on linux.
Singularity> pip install --upgrade "jax[cuda12_pip]" -f https://
storage.googleapis.com/jax-releases/jax_cuda_releases.html

# Other packages
Singularity> pip3 install jupyter jupyterhub pandas matplotlib
```

We can verify that there is still room on our image:

```
Singularity> find /ext3 | wc -l
# 59867
Singularity> du -sh  /ext3
# 4.1G    /ext3
```

Lastly, I exit the `Singularity` container and rename the `.ext3` image into something more appropriate:

```
Singularity> exit
$ mv overlay-15GB-500K.ext3 my_jax.ext3
```

The `my_jax.ext3` image contains everything to run `JAX` on the cluster. We can now access `Singularity` and activate the environment using

```
$ singularity exec --overlay my_jax.ext3:rw /scratch/work/public/singularity/
cuda12.1.1-cudnn8.9.0-devel-ubuntu22.04.2.sif /bin/bash
Singularity> source /ext3/env.sh
Singularity> conda activate recoveredenv #running the virtual environment
recoveredenv
```

6. Running `JAX`

To run a `Python` script on the cluster, I include the necessary `#!/bin/env python` line of code called a `shebang`.

In this document, I run the `jax_detect_GPU.py` file as a test:

```
#!/bin/env python

from jax.lib import xla_bridge

if __name__ == "__main__":
    xla_backend = xla_bridge.get_backend()
    xla_backend_type = xla_bridge.get_backend().platform  # cpu, gpu, tpu
    print(f"XLA backend type: {xla_backend_type}")

    gpu_count = xla_backend.device_count() if xla_backend_type == "gpu" else 0
    print(f"\nNumber of GPUs found on system: {gpu_count}")
    if xla_backend_type == "gpu":
        for idx, device in enumerate(xla_backend.devices()):
            gpu_type = "Active GPU" if idx == 0 else "GPU"
            print(f"\n{gpu_type} index: {device.id}")
            print(f"{gpu_type} name: {device.device_kind}")
```

I want to run `jax_detect_GPU.py` but I need to use a `SLURM` script. The `SLURM` script called `run-test.SBATCH` has contents:

```
#!/bin/bash

#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --time=1:00:00
#SBATCH --mem=2GB
#SBATCH --gres=gpu
#SBATCH --job-name=jax

module purge

singularity exec --nv \
    --overlay /scratch/vv2102/jax-example/my_jax.ext3:ro \
    /scratch/work/public/singularity/cuda12.1.1-cudnn8.9.0-devel-ubuntu22.04.2.
sif\
    /bin/bash -c "source /ext3/env.sh; python jax_detect_GPU.py"
```

Finally, I execute the script:

```
$ sbatch run-test.SBATCH
```

I can verify that `JAX` has worked by inspecting the file contents of the `slurm-xxxxxxx.out` file:

```
$ nano slurm-36537735.out

# XLA backend type: gpu
# Number of GPUs found on system: 1
# Active GPU index: 0
# Active GPU name: Tesla V100-PCIE-32GB
```

The file confirms `JAX` is running on GPU `Tesla V100-PCIE-32GB`.

Section E: Running custom environments on the Greene cluster

Julia on Greene Cluster

Create a directory for your julia work, such as /scratch//julia, and then change to your home directory. An example is shown below.

```
mkdir /home/<NetID>/julia
cd /home/<NetID>/julia'
```

Copy an overlay image, such as the 2GB 100K overlay, which generally has enough storage for Julia packages. Once copied, unzip to the same folder, rename to julia-pkgs.ext3

```
cp -rp /scratch/work/public/overlay-fs-ext3/overlay-2GB-100K.ext3.gz .
gunzip overlay-2GB-100K.ext3.gz
mv overlay-2GB-100K.ext3 julia-pkgs.ext3
```

Copy the following wrapper script in the directory

```
cp -rp /share/apps/utils/julia-setup/* .
```

Now launch writable Singularity overlay, run the Julia REPL, and install packages

```
srun --cpus-per-task=2 --mem=10GB --time=04:00:00 --pty /bin/bash
module purge
bash my-julia-writable
```