NERSC User's Group (NUG) Community Call:

Using CUDA with Python on Perlmutter@NERSC

15 August, 2024

Today's Pipeline

- Front Matter/Getting Started
- Jupyter Primer
- Interactive Jupyter Notebook
    - Python Review
    - Cuda with Python
- Introduction to CuPy
- Advanced Features of CuPy

Some Logistics

- In-person attendees please also join Zoom for full participation
- Please change your name in Zoom session
  - to: first_name last_name
  - Click “Participants”, then “More” next to your name to rename
- Click the CC button to toggle captions and View Full Transcript
- Session is being recorded
- Users are muted upon joining Zoom
  - Feel free to unmute and ask questions or ask in GDoc below
- GDoc is used for Q&A (instead of Zoom chat)
- Please answer a short survey afterward

Some Logistics

- Slides and videos will be available on Grads@NERSC page and NERSC Training Event page
- Introduction to CUDA Programming Training
  - https://www.nersc.gov/users/training/training-materials/
    - previous training materials available
    - 13-Part Detailed CUDA Training

Hands-on Exercises on Perlmutter

ssh <user>@perlmutter.nersc.gov

- `% cd $SCRATCH`
- `% git clone` https://github.com/charlesWLivelyPhD/NERSC_CUDAPYTHON_August2024
  - Downloads all exercises (and answers!)
- References
  - https://docs.nersc.gov/jobs/
  - https://docs.nersc.gov/jobs/examples/#interactive

Using Perlmutter Compute Node Reservations

- Existing NERSC users (at time of registration) have been added to "`ntrain3`" project
- Non-NERSC users have received email instructions on apply for a training account
    - Please let us know if you need one
- Perlmutter node reservations: 10:30 am - 4:30 pm PDT today
    - `--reservation=august_nug -A ntrain3 -C gpu`

      `(add -q shared -c 32 -G 1 for shared)`

      for sbatch or salloc sessions
    - No need to use `--reservation` or `-A` when outside of the reservation hours

NERSC Code of Conduct

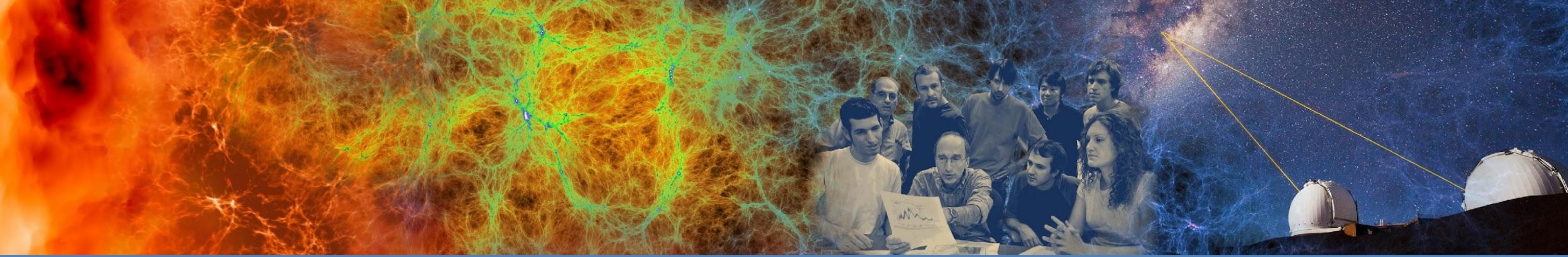
- Team Science
- Service
- Trust
- Innovation
- Respect

- We agree to **work together professionally and productively** towards our shared goals while respecting each other's differences and ideas.

- We should all feel free to speak up to maintain this environment and remember there are resources available to **report violations** to foster an inclusive, collaborative environment.
  Email nersc-training@lbl.gov for any concerns

https://www.nersc.gov/nersc-code-of-conduct

NERSC U.S. DEPARTMENT OF ENERGY | Office of Science BERKELEY LAB Bringing Science Solutions to the World U.S. DEPARTMENT OF ENERGY | Office of Science

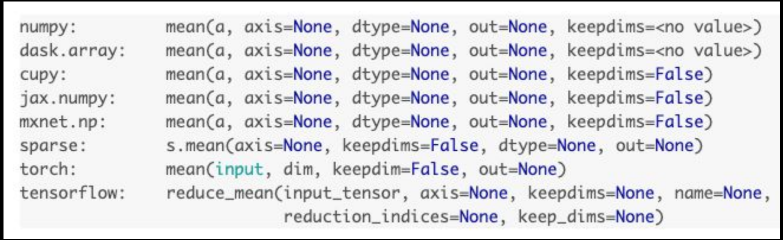


Using Python on GPUs and Jupyter on Perlmutter@NERSC

Getting started with GPUs in Python

- NumPy and SciPy do not utilize GPUs out of the box

- There are many Python GPU frameworks out there:
  - “drop in” replacements for numpy, scipy, pandas, scikit-learn, etc
    - **CuPy**, **RAPIDS**
  - “machine learning” libraries that also support general GPU computing
    - **PyTorch**, **TensorFlow, JAX**
  - “I want to write my own GPU kernels”
    - **Numba, CUDA Python**
  - multi-gpu / multi-node / distributed memory:
    - **mpi4py+X, dask, cuNumeric**
- Many of these GPU libraries have adopted the CUDA Array Interface which makes it easier to pass array-like objects stored in GPU memory between the libraries
- There is also effort in the community to standardize around a common Python array API



```
numpy:        mean(a, axis=None, dtype=None, out=None, keepdims=<no value>)
dask.array:   mean(a, axis=None, dtype=None, out=None, keepdims=<no value>)
cupy:         mean(a, axis=None, dtype=None, out=None, keepdims=False)
jax.numpy:    mean(a, axis=None, dtype=None, out=None, keepdims=False)
mxnet.np:     mean(a, axis=None, dtype=None, out=None, keepdims=False)
sparse:       s.mean(axis=None, keepdims=False, dtype=None, out=None)
torch:        mean(input, dim, keepdim=False, out=None)
tensorflow:   reduce_mean(input_tensor, axis=None, keepdims=None, name=None,
                          reduction_indices=None, keep_dims=None)
```

cudatoolkit dependency via module

```
> module load conda
```

Note: cudatoolkit module is loaded by default
Current default version is cudatoolkit/12.2

```
> conda create --name cupy-demo python=3.12 numpy scipy
> conda activate cupy-demo
> pip install cupy-cuda11X
> python
>>> import cupy as cp
>>> print(cp.array([1, 2, 3]))
[1 2 3]
```

Check your package documentation to see cudatoolkit compatibility requirements

https://docs.nersc.gov/development/languages/python/using-python-perlmutter/

cudatoolkit dependency via conda

cupy conda-forge package will pull cudatoolkit dependencies into conda env

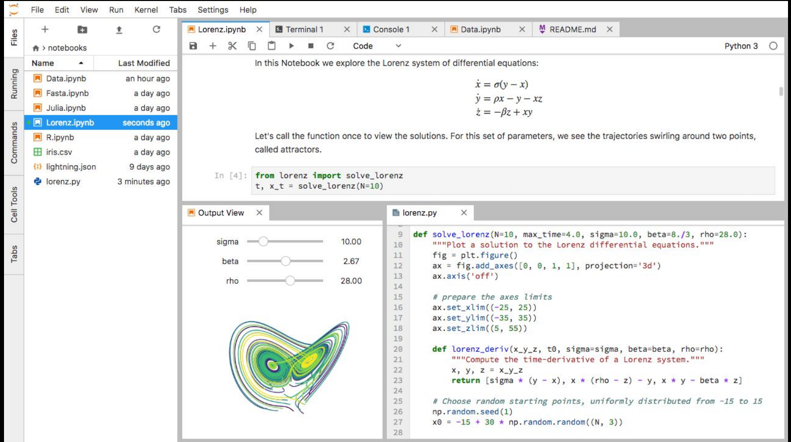
```
> module load conda
> module unload cudatoolkit
> conda create --name cupy-demo python=3.11 numpy scipy
> conda activate cupy-demo
> conda install -c conda-forge cupy
> python
>>> import cupy as cp
>>> print(cp.array([1, 2, 3]))
[1 2 3]
```

cupy conda-forge package will pull cudatoolkit dependencies into conda env

https://docs.nersc.gov/development/languages/python/using-python-perlmutter/

What is Jupyter?

- At NERSC, we say “Jupyter” in reference to a collection of many things
  - Access shareable Jupyter “notebooks” via JupyterHub
- What can I put in a Jupyter notebook?
  - Live code
  - Equations
  - Visualizations
  - Narrative text
  - Interactive widgets
- What applications would I use a notebook for?
  - Data cleaning and data transformation
  - Numerical simulation
  - Statistical modeling
  - Data visualization
  - Machine learning
  - Workflows and analytics frameworks



Name	Last Modified
Data.ipynb	an hour ago
Fasta.ipynb	a day ago
Julia.ipynb	a day ago
Lorenz.ipynb	seconds ago
R.ipynb	a day ago
iris.csv	a day ago
lightning.json	9 days ago
lorenz.py	3 minutes ago

In this Notebook we explore the Lorenz system of differential equations:

$$\dot{x} = \sigma(y - x)$$
$$\dot{y} = \rho x - y - xz$$
$$\dot{z} = -\beta z + xy$$

Let's call the function once to view the solutions. For this set of parameters, we see the trajectories swirling around two points, called attractors.

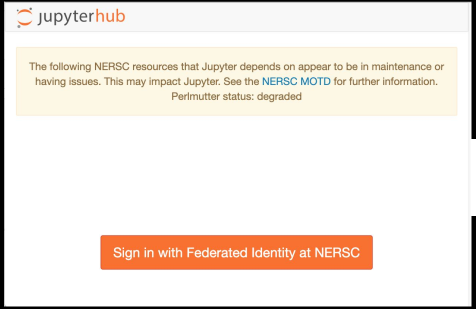
```
In [4]: from lorenz import solve_lorenz
        t, x_t = solve_lorenz(N=10)
```

```
sigma   10.00
beta    2.67
rho     28.00
```

```python
def solve_lorenz(N=10, max_time=4.0, sigma=10.0, beta=8./3, rho=28.0):
    """Plot a solution to the Lorenz differential equations."""
    fig = plt.figure()
    ax = fig.add_axes([0, 0, 1, 1], projection='3d')
    ax.axis('off')

    # prepare the axes limits
    ax.set_xlim((-25, 25))
    ax.set_ylim((-35, 35))
    ax.set_zlim((5, 55))

    def lorenz_deriv(x_y_z, t0, sigma=sigma, beta=beta, rho=rho):
        """Compute the time-derivative of a Lorenz system."""
        x, y, z = x_y_z
        return [sigma * (y - x), x * (rho - z) - y, x * y - beta * z]

    # Choose random starting points, uniformly distributed from -15 to 15
    np.random.seed(1)
    x0 = -15 + 30 * np.random.random((N, 3))

```

How Do I Use Jupyter at NERSC?

- https://jupyter.nersc.gov



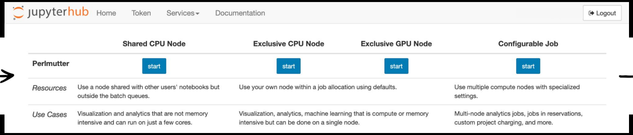
jupyterhub

The following NERSC resources that Jupyter depends on appear to be in maintenance or having issues. This may impact Jupyter. See the NERSC MOTD for further information.
Perlmutter status: degraded

Sign in with Federated Identity at NERSC

Authenticate

$\longrightarrow$

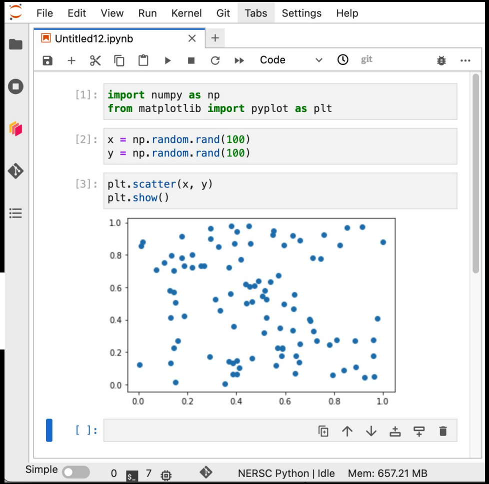


jupyterhub Home Token Services▾ Documentation Logout

	Shared CPU Node	Exclusive CPU Node	Exclusive GPU Node	Configurable Job
Perlmutter	start	start	start	start
Resources	Use a node shared with other users' notebooks but outside the batch queues.	Use your own node within a job allocation using defaults.		Use multiple compute nodes with specialized settings.
Use Cases	Visualization and analytics that are not memory intensive and can run on just a few cores.	Visualization, analytics, machine learning that is compute or memory intensive but can be done on a single node.		Multi-node analytics jobs, jobs in reservations, custom project charging, and more.

Choose

$\longrightarrow$



```
[1]: import numpy as np
     from matplotlib import pyplot as plt

[2]: x = np.random.rand(100)
     y = np.random.rand(100)

[3]: plt.scatter(x, y)
     plt.show()
```

```
[ ]:
```

Simple 0 7 NERSC Python | Idle Mem: 657.21 MB

Go!

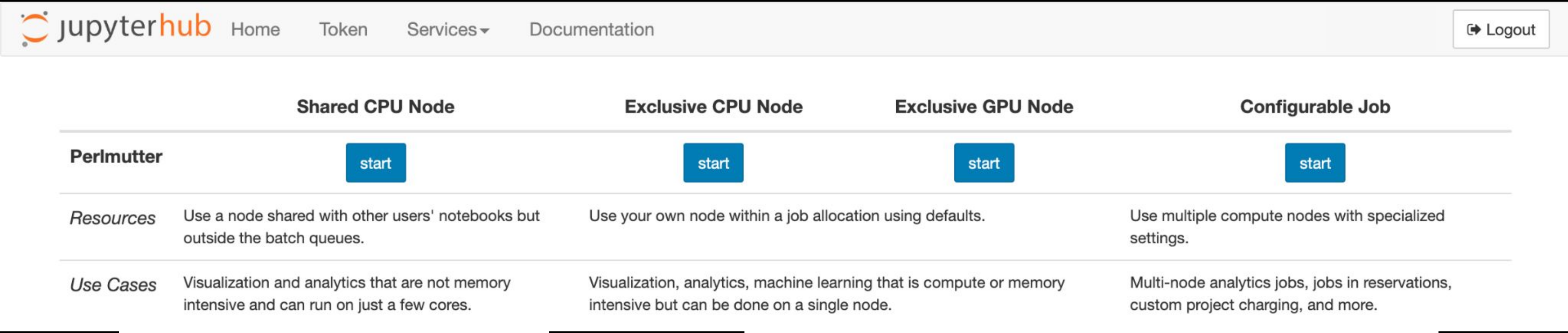
How Do I Choose a Notebook Server to Spawn?

Shared CPU:

Exclusive CPU/GPU:

Configurable Job:

!sbatch

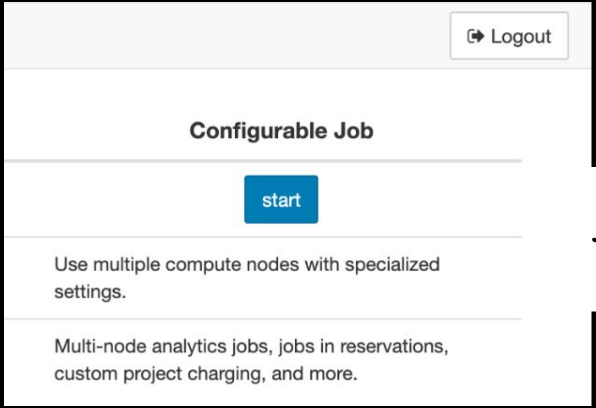


	Shared CPU Node	Exclusive CPU Node	Exclusive GPU Node	Configurable Job
Perlmutter	start	start	start	start
Resources	Use a node shared with other users' notebooks but outside the batch queues.	Use your own node within a job allocation using defaults.		Use multiple compute nodes with specialized settings.
Use Cases	Visualization and analytics that are not memory intensive and can run on just a few cores.	Visualization, analytics, machine learning that is compute or memory intensive but can be done on a single node.		Multi-node analytics jobs, jobs in reservations, custom project charging, and more.

Shared = other users and processes on the same node

Exclusive and configurable = compute nodes just for your notebook and processes

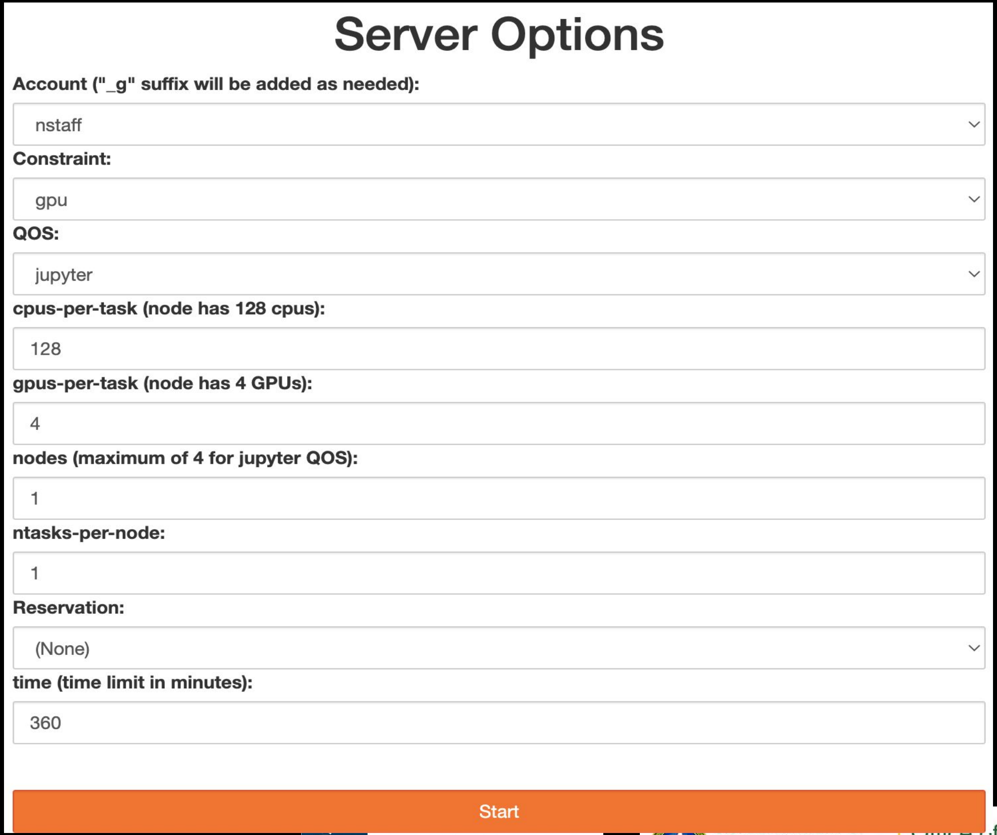
Configurable Job Settings



Configurable Job

Use multiple compute nodes with specialized settings.

Multi-node analytics jobs, jobs in reservations, custom project charging, and more.


Server Options

Account ("_g" suffix will be added as needed): nstaff

Constraint: gpu

QOS: jupyter

cpus-per-task (node has 128 cpus): 128

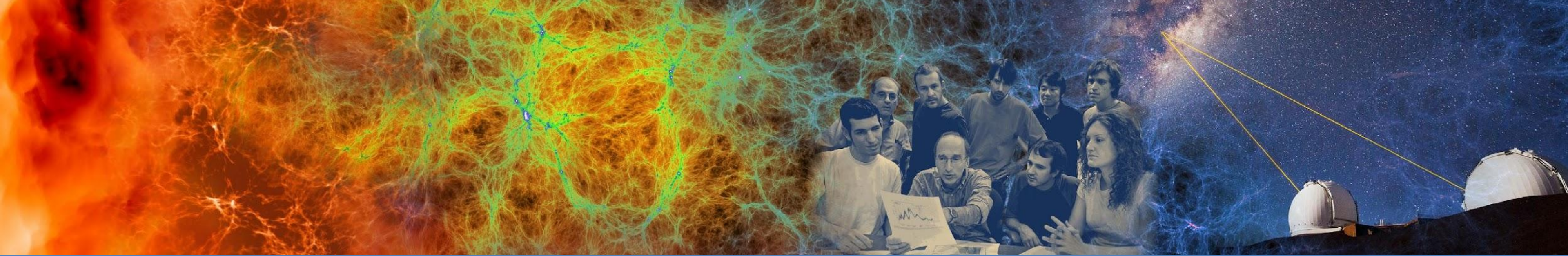
gpus-per-task (node has 4 GPUs): 4

nodes (maximum of 4 for jupyter QOS): 1

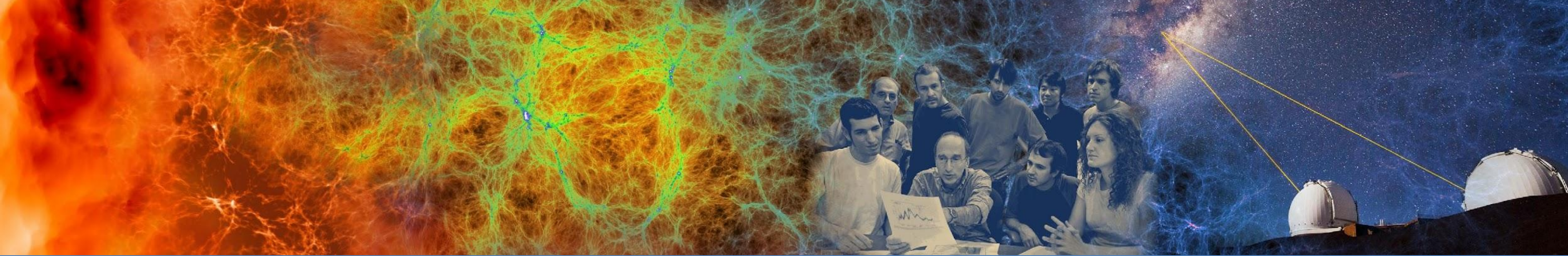
ntasks-per-node: 1

Reservation: (None)

time (time limit in minutes): 360



Launch Interactive Jupyter Notebook



Introduction to CuPY

CuPy is...

a library to provide NumPy-compatible features with GPU

NumPy

```
import numpy as np
X_cpu = np.zeros((10,))
W_cpu = np.zeros((10, 5))
y_cpu = np.dot(x_cpu,
W_cpu)
```

CuPy

```
import cupy as cp
x_gpu = cp.zeros((10,))
W_gpu = cp.zeros((10, 5))
y_gpu = cp.dot(x_gpu,
W_gpu)
```

```
y_cpu = cp.asnumpy(y_gpu)
```

```
y_gpu = cp.asarray(y_cpu)
```

Comparing NumPy and CuPy

```
import numpy as np
X_cpu =
np.zeros((10,))
W_cpu = np.zeros((10, 5))
y_cpu = np.dot(x_cpu, W_cpu)
```

```
import cupy as cp
x_gpu = cp.zeros((10,))
W_gpu = cp.zeros((10, 5))
y_gpu = cp.dot(x_gpu,
W_gpu)
```

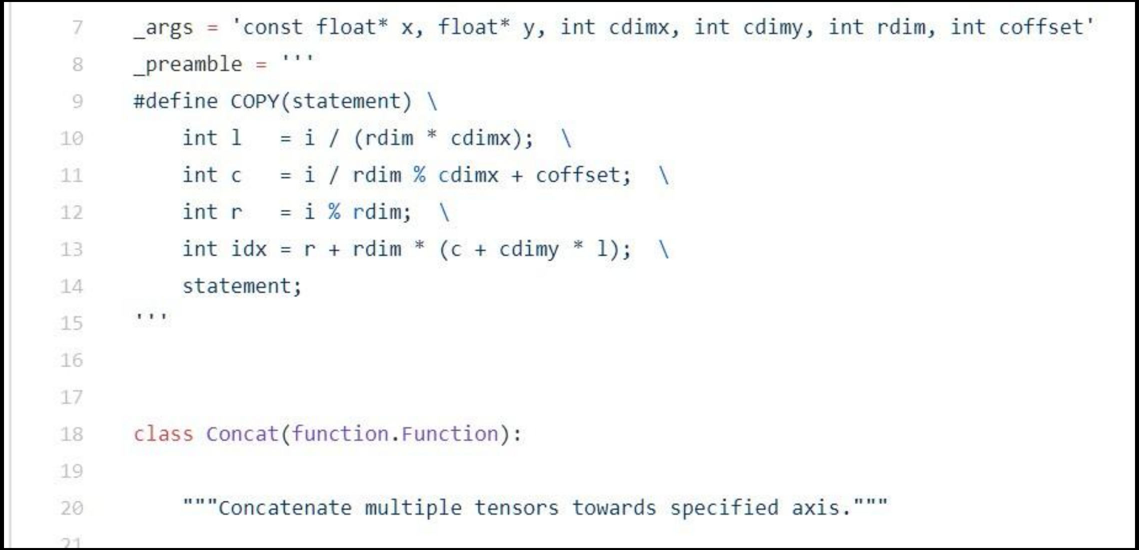
```
for xp in [np, cp]:
    x = xp.zeros((10,))
    W = xp.zeros((10, 5))
    y = xp.dot(x, W)
```

Support both CPU and GPU with the same code!

Why develop CuPy? (1)

- Chainer functions had separate implementations in NumPy and PyCUDA to support both CPU and GPU

Even writing simple functions like "Add" or "Concat" took several lines...



```
_args = 'const float* x, float* y, int cdimx, int cdimy, int rdim, int coffset'
_preamble = '''
#define COPY(statement) \
    int l   = i / (rdim * cdimx);  \
    int c   = i / rdim % cdimx + coffset;  \
    int r   = i % rdim;  \
    int idx = r + rdim * (c + cdimy * l);  \
    statement;
'''


class Concat(function.Function):

    """Concatenate multiple tensors towards specified axis."""
```

Why develop CuPy? (2)

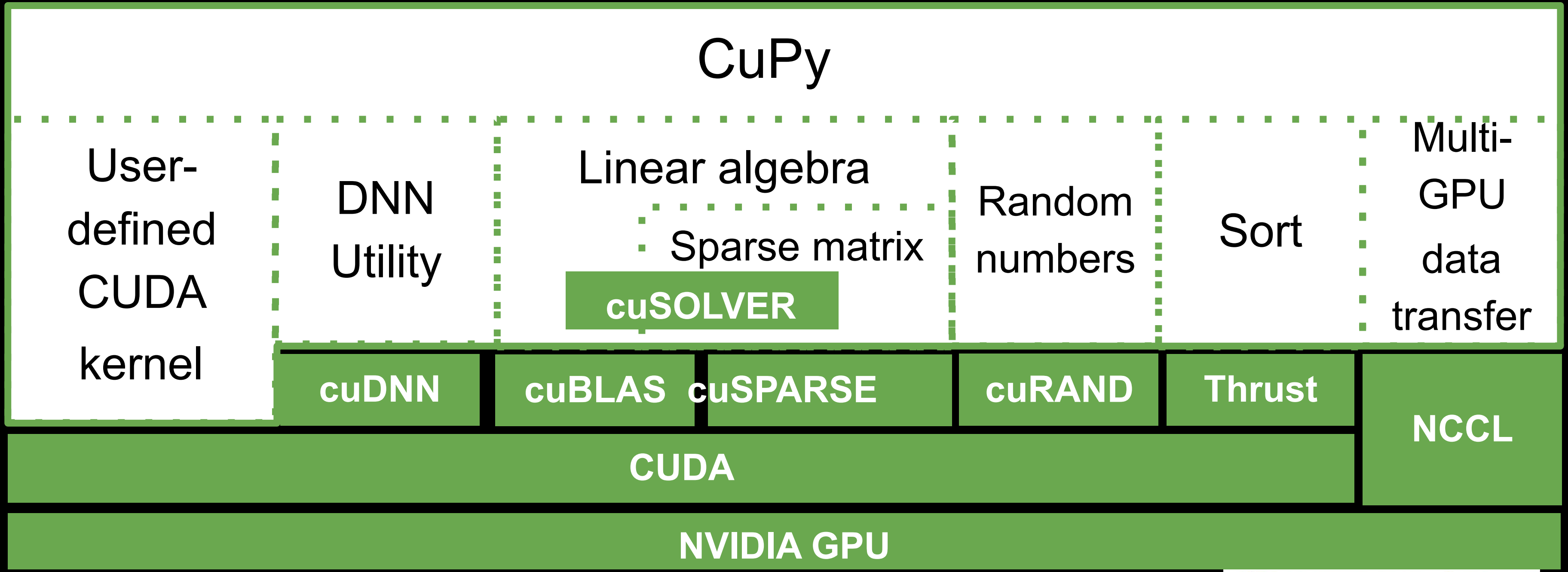
- Needed a NumPy-compatible GPU array library
  - NumPy is complicated
    - dtypes
    - Broadcast
    - Indexing

Why develop CuPy? (3)

- There was no convenient library
  - gnumpy
    - Consists of a single file which has 1000 lines of code
    - Not currently maintained
  - CUDA-based NumPy
    - No pip package is provided

⇒NVIDIA Needed to develop it

Inside CuPy

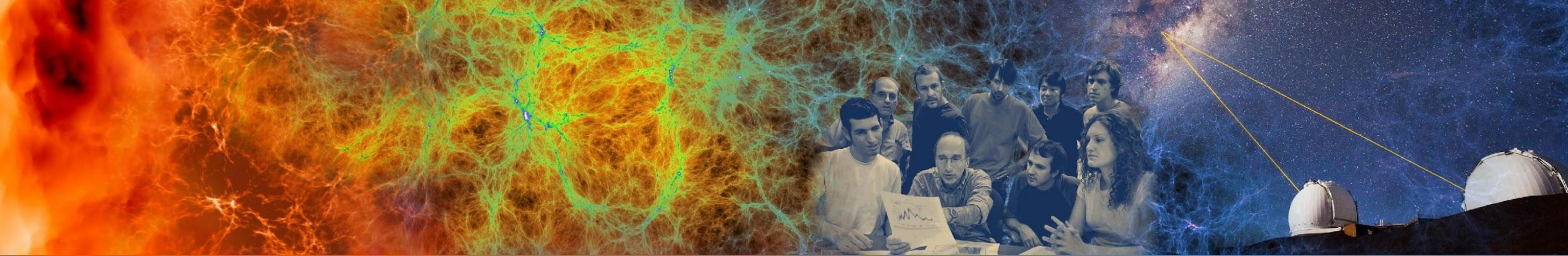

CuPy
User-defined CUDA kernel
DNN Utility
Linear algebra
Sparse matrix
cuSOLVER
Random numbers
Sort
Multi-GPU data transfer
cuDNN
cuBLAS
cuSPARSE
cuRAND
Thrust
NCCL
CUDA
NVIDIA GPU

NumPy compatible features

- Data types (dtypes)
  - bool_, int8, int16, int32, int64, uint8, uint16, uint32, uint64, float16, float32, float64, complex64, and complex128
- All basic indexing
  - indexing by ints, slices, newaxes, and Ellipsis
- Most of advanced indexing
  - except indexing patterns with boolean masks
- Most of the array creation routines
  - empty, ones_like, diag, etc...
- Most of the array manipulation routines
  - reshape, rollaxis, concatenate, etc...
- All operators with broadcasting
- All universal functions for element-wise operations
  - except those for complex numbers
- Linear algebra functions accelerated by cuBLAS
  - including product: dot, matmul, etc...
  - including decomposition: cholesky, svd, etc...
- Reduction along axes
  - sum, max, argmax, etc...
- Sort operations implemented by Thrust
  - sort, argsort, and lexsort
- Sparse matrix accelerated by cuSPARSE

New features after CuPy v2

- Narrowed the gap with NumPy
- Speedup: Cythonized, Improved MemoryPool
- CUDA Stream support
- Added supported functions
  - From NumPy
  - Sparse Matrix, FFT, scipy ndimage support



Advanced Features in CuPY

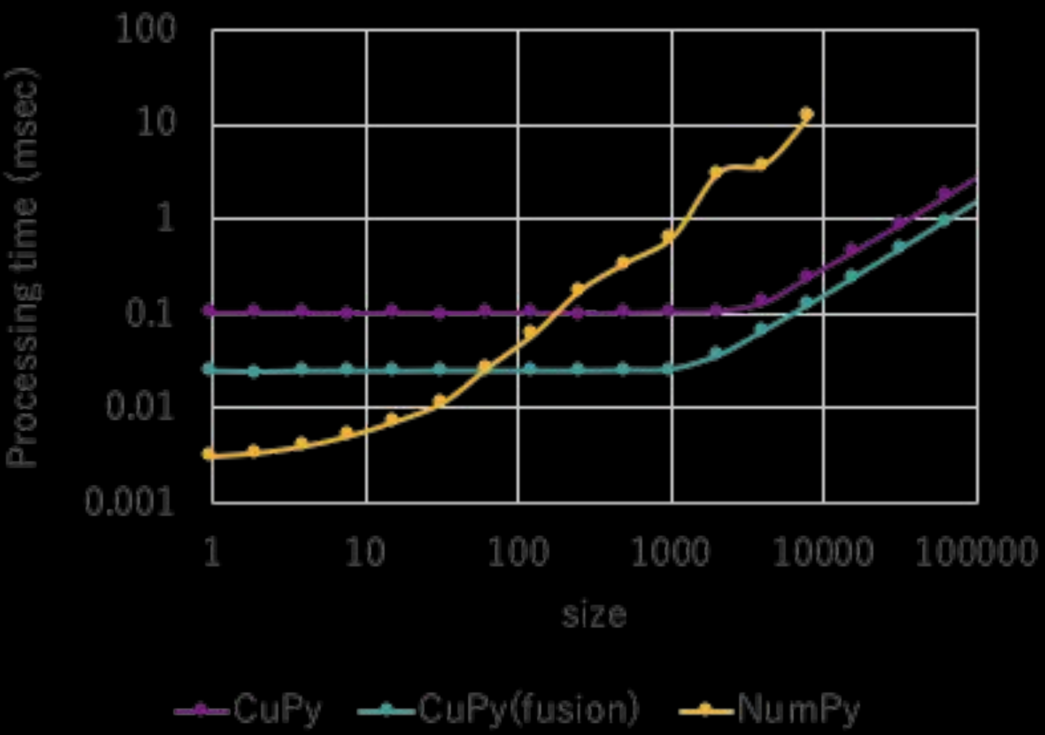
Overview of Advanced Features

- Kernel Fusion
- Unified Memory
- Custom Kernels
- Compatibility with other libraries
    - SciPy-compatible features
    - Direct use of NumPy functions via array_interface
    - Numba
    - PyTorch via DLPack
    - cuDF / cuML

Fusion: fuse kernels for further speedup!

```python
a = numpy.float32(2.0)
x = xp.ones((1024, size), 'f')
y = xp.ones((1024, size), 'f')

def saxpy(a, x, y):
    return a * x + y


saxpy(a, x, y)  # target
```

```python
@cupy.fuse()
def saxpy(a, x, y):
    return a * x + y


saxpy(a, x, y)  # target
```



- Speedup function calls
- Reduce memory consumption
- Relax the bandwidth bottleneck

Limitations of @cupy.fuse()

- Only element-wise and reduction operations are supported
- Other operations like `cupy.matmul()` and `cupy.reshape()` are
  not yet supported

You want to save GPU memory?

```python
import cupy as cp
size = 32768
a = cp.ones((size, size)) # 8GB
b = cp.ones((size, size)) # 8GB
cp.dot(a, b)   # 8GB
```

```
Traceback (most recent call last):
  ...
cupy.cuda.memory.OutOfMemoryError: out of memory
to allocate 8589934592 bytes (total 17179869184
bytes)
```

Try Unified Memory! (Supported only on V100)

- Just edit 2 lines to enable **unified memory**

```
import cupy as cp

pool =
cp.cuda.MemoryPool(cp.cuda.malloc_managed)
cp.cuda.set_allocator(pool.malloc)
size = 32768
a = cp.ones((size, size)) # 8GB
b = cp.ones((size, size)) # 8GB
cp.dot(a, b)              # 8GB
```

- CuPy provides classes to compile **your own CUDA kernel**:
  - ElementwiseKernel
  - ReductionKernel
  - **RawKernel(from v5)**
    - For CUDA experts who love to write everything by themselves
    - Compiled with NVRTC

Basic usage of ElementwiseKernel

```
squared_diff = cp.ElementwiseKernel(
    'float32 x, float32 y',    # input params
    'float32 z',               # output params
    'z = (x - y) * (x - y)',   # element-wise operation
    'squared_diff'             # the name of this kernel
)

x = cp.arange(10, dtype=np.float32).reshape(2, 5)
y = cp.arange(5, dtype=np.float32)

squared_diff(x, y)
```

Type-generic kernels

```
squared_diff_generic =
cp.ElementwiseKernel(
    'T x, T y',                 # input params
    'T z',                      # output params
    'z = (x - y) * (x - y)',    # element-wise operation
    'squared_diff'              # the name of this
)                               kernel

x = cp.arange(10, dtype=np.float32).reshape(2,
5) y = cp.arange(5, dtype=np.float32)

squared_diff_generic(x, y)
```

Type-generic kernels

```
squared_diff_generic = cp.ElementwiseKernel(
    'T x, T y',
    'T z',
    '''
        T diff = x - y; z =
        diff * diff;
    ''',
    'squared_diff_generic')

x = cp.arange(10, dtype=np.float32).reshape(2, 5)
y = cp.arange(5, dtype=np.float32)

squared_diff_generic(x, y)
```

Manual indexing with raw

```
add_reverse = cp.ElementwiseKernel(
    'T x, raw T y',                    # input params
    'T z',                             # output params
    'z = x + y[_ind.size() - i - 1]',  # element-wise operation
    'add_reverse'                      # the name of this kernel
)

x = cp.arange(5,
dtype=np.float32) y =
cp.arange(5, dtype=np.float32)


add_reverse(x, y)
```

```
x + y[::-1]
```

Reduction Kernel

```
l2norm_kernel = cp.ReductionKernel(
     'T x',              # input array
     'T y',              # output array
    'x * x',            # map
    'a + b',            # reduce
    'y = sqrt(a)',      # post-reduction map
    '0',                # identity value
    'l2norm'            # kernel name
)
x = cp.arange(1000, dtype=np.float32).reshape(20,
50) l2norm_kernel(x, axis=1)
```

=> This is same as : `cp.sqrt((x * x).sum(axis=1))`
but much faster!

NERSC U.S. DEPARTMENT OF ENERGY Office of Science BERKELEY LAB Bringing Science Solutions to the World U.S. DEPARTMENT OF ENERGY Office of Science

How a RawKernel looks...

```
import cupy as cp

square_kernel = cp.RawKernel(r'''
extern "C" __global   void my_square(long long* x) {
    int tid = threadIdx.x;
    x[tid] *= x[tid];
}
''', name='my_square')

x = cp.arange(5)
square_kernel(grid=(1,), block=(5,), args=(x,))
print(x)   # [ 0   1   4   9 16]
```

Now onto the Interactive Notebooks