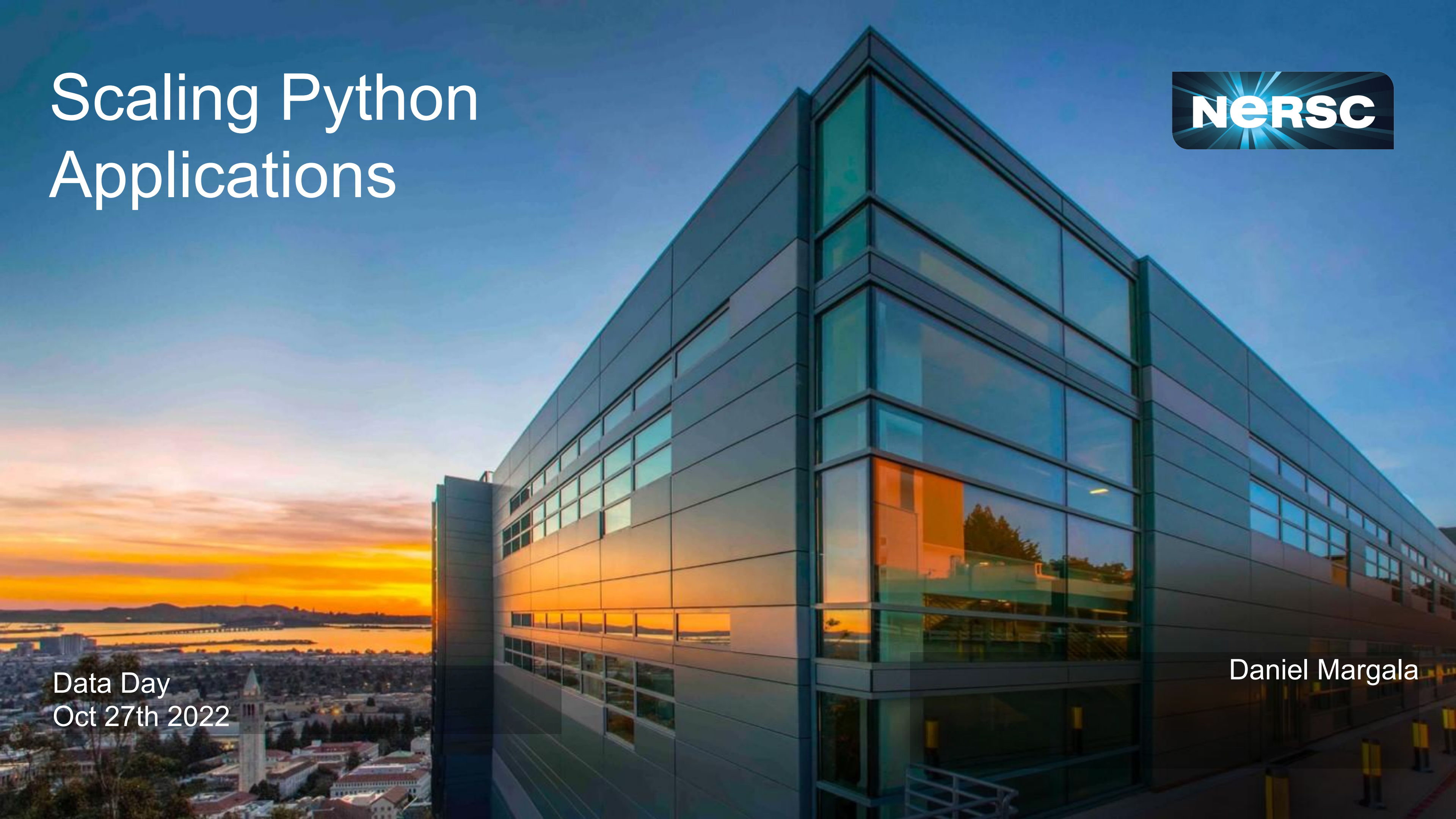


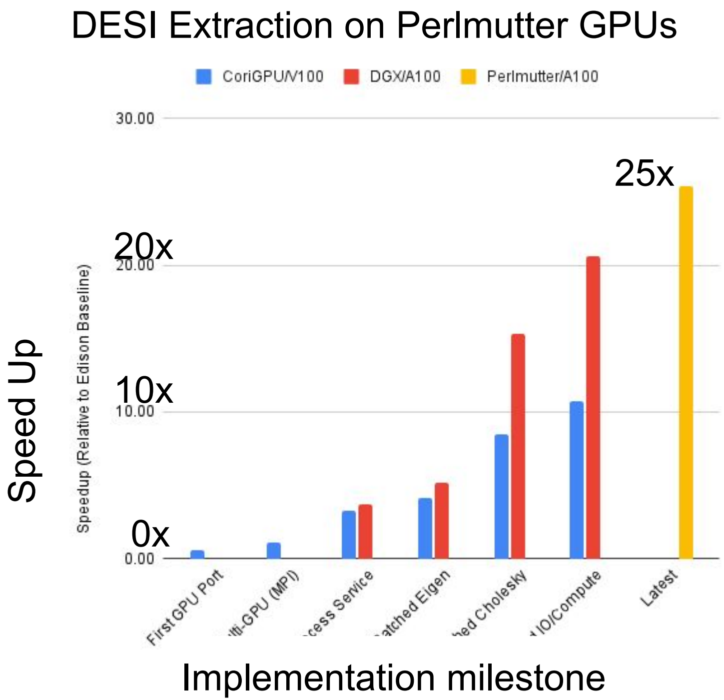
Scaling Python Applications

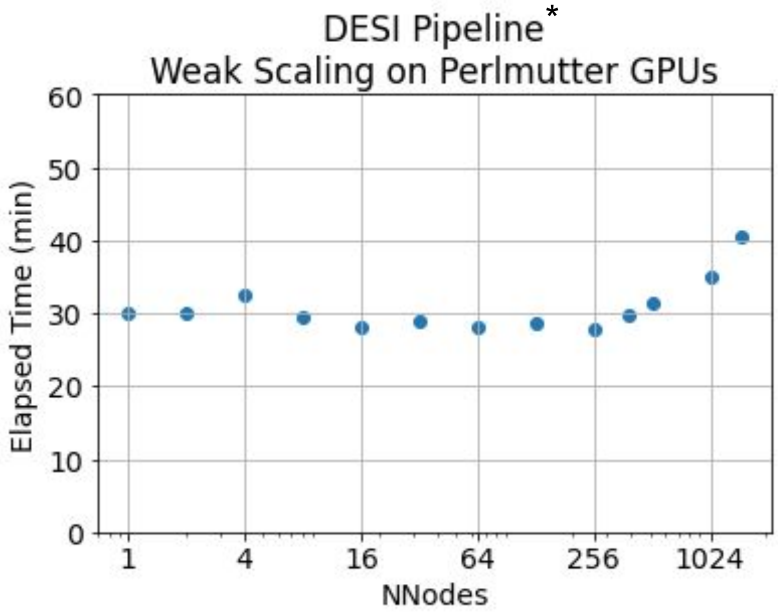
NERSC

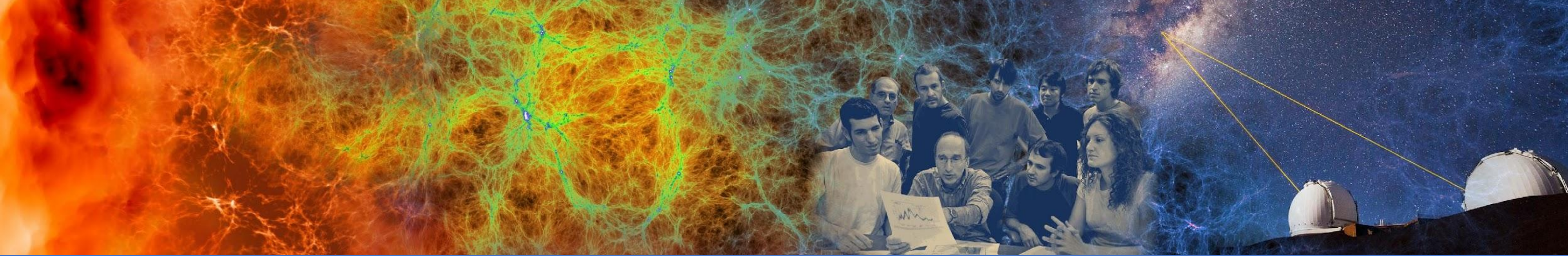
Data Day
Oct 27th 2022

Daniel Margala

DESI Data Processing on Perlmutter







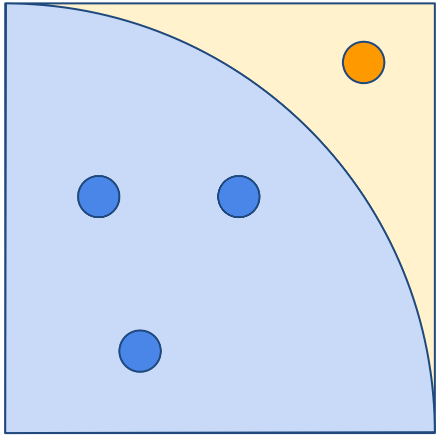
Parallelism in Python

Example problem: Monte Carlo Pi

library.py

```
import random

def estimate_pi(n):
    c = 0
    for i in range(n):
        x = random.uniform(0, 1)
        y = random.uniform(0, 1)
        if x*x + y*y < 1:
            c += 1
    return c * 4.0 / n
```



Some terms

A **program** is a collection of instructions for a computer to execute.

A **process** is an instance of a program that is being executed. Contains one or more threads.

A **thread** is a unit of execution within a process. Typically, multiple threads within a process share process state and memory.

Serial Python

pi-serial.py

```
import time
from library import estimate_pi


n = 20_000_000

start = time.time()
result = estimate_pi(n)
end = time.time()

print(end - start)
```

```
> python pi-serial.py
3.6154
```

The python interpreter transforms the code into Python bytecode instructions and then executes those instructions at runtime.

Python is slower than compiled languages like c, c++, fortran but developers like it for productivity and ease of use.

A simple c version of this example is about an order of magnitude faster than the Python version.

Python 3.11.0!

Release Date: Oct. 24, 2022 https://www.python.org/downloads/release/python-3110/

General changes

- PEP 657 -- Include Fine-Grained Error Locations in Tracebacks
- PEP 654 -- Exception Groups and except*
- PEP 680 -- tomllib: Support for Parsing TOML in the Standard Library
- gh-90908 -- Introduce task groups to asyncio
- gh-34627 -- Atomic grouping ((?>...)) and possessive quantifiers (*+, ++, ?+, {m,n}+) are now supported in regular expressions.
- The Faster CPython Project is already yielding some exciting results. **Python 3.11 is up to 10-60% faster than Python 3.10. On average, we measured a 1.22x speedup on the standard benchmark suite. See Faster CPython for details.**

Serial Python (redux)

pi-serial.py

```
import time
from library import estimate_pi


n = 20_000_000

start = time.time()
result = estimate_pi(n)
end = time.time()

print(end - start)
```

```
(py310) > python pi-serial.py
3.17

(py311) > python pi-serial.py
2.21
```

"free speedup is the best speedup"
-Laurie Stephey

Multithreading in Python

pi-threads.py

```
import time
from library import estimate_pi
from threading import Thread

n = 20_000_000
p = 4

threads = [
    Thread(target=estimate_pi, args=(n//p,))
    for i in range(p)
]

start = time.time()
[t.start() for t in threads]
[t.join() for t in threads]
end = time.time()

print(end - start)
```

```
> python pi-threads.py
3.8097
```

The Global Interpreter Lock (GIL) in Python prevents compute-bound threads from making progress in parallel.

For the most part, don't bother with multithreading for scientific data processing in Python

Shared memory
Low overhead for starting up threads

Multithreading in Python

sleep-serial.py

```
import time

def task(n):
    time.sleep(n)

n = 5

start = time.time()
task(n)
end = time.time()

print(end - start)
```

```
> python sleep-serial.py
5.0050
```

sleep-threads.py

```
import time
from threading import Thread
def task(n):
    time.sleep(n)
n = 5
p = 4
t = [
    Thread(target=task, args=(n/p,))
    for i in range(p)
]
start = time.time()
[t[i].start() for i in range(p)]
[t[i].join() for i in range(p)]
end = time.time()
print(end - start)
```

```
> python sleep-threads.py
1.2515
```

Python 3.12!

https://github.com/faster-cpython/ideas/wiki/Python-3.12-Goals

Multi-threaded parallelism

Python currently has a single global interpreter lock per process, which prevents multi-threaded parallelism. This work, described in PEP 684, is to make all global state thread safe and move to a global interpreter lock (GIL) per sub-interpreter. Additionally, PEP 554 will make it possible to create subinterpreters from Python (currently a C API-only feature), opening up true multi-threaded parallelism.

Multiprocessing in Python

pi-multiprocessing.py

```
import time
from library import estimate_pi
import multiprocessing as mp

n = 20_000_000
p = 4

if __name__ == "__main__":

    mp.set_start_method("spawn")
    start = time.time()
    with mp.Pool(processes=p) as pool:
        results = pool.map(estimate_pi, [n//p]*p)
    end = time.time()
    print(end - start)
```

```
> python pi-multiprocessing.py
1.0609
```

“spawn”: parent process starts a fresh Python interpreter process. The child process will only inherit those resources necessary to run the process object’s run() method.

“fork”: child process is identical to parent process, all resources are inherited.

More overhead than using threads.
Distributed memory
Limited to single node.

MPI Parallelism in Python

The Message-Passing Interface (MPI) is a set of library functions which are used to facilitate inter-process communication on parallel computing systems.

Popular open source implementations of MPI are MPICH and OpenMPI. The officially supported implementation at NERSC is cray-mpich.

mpi4py builds on the MPI specification and provides a Python interface to standard MPI functions. It supports point-to-point and collective communication of buffer objects (such as NumPy arrays) and picklable Python objects

MPI Parallelism in Python

pi-mpi.py

```
import time
from library import estimate_pi

n = 20_000_000

if __name__ == "__main__":

    from mpi4py import MPI
    comm = MPI.COMM_WORLD
    p = comm.size

    comm.barrier(); start = time.time()
    result = estimate_pi(n//p)
    comm.barrier(); end = time.time()

    if comm.rank == 0:
        print(end - start)
```

```
> srun -n 4 python pi-mpi.py
0.9922
```

MPI launcher is responsible for launching processes. Processes sync up during initialization (from mpi4py import MPI)

Standard communication semantics help with move data movement and coordination between process.

Very popular framework in HPC.
Distributed memory.
Can scale out to multiple nodes!

Parallelism with Dask

pi-dask.py

```
import time
from library import estimate_pi
import dask
from dask.distributed import Client, progress

n = 20_000_000
p = 4
if __name__ == "__main__":
    client = Client(threads_per_worker=1, n_workers=p)
    futures = [
        dask.delayed(estimate_pi)(n//p)
        for i in range(p)
    ]
    start = time.time()
    dask.compute(*futures)
    end = time.time()
    print(end - start)
```

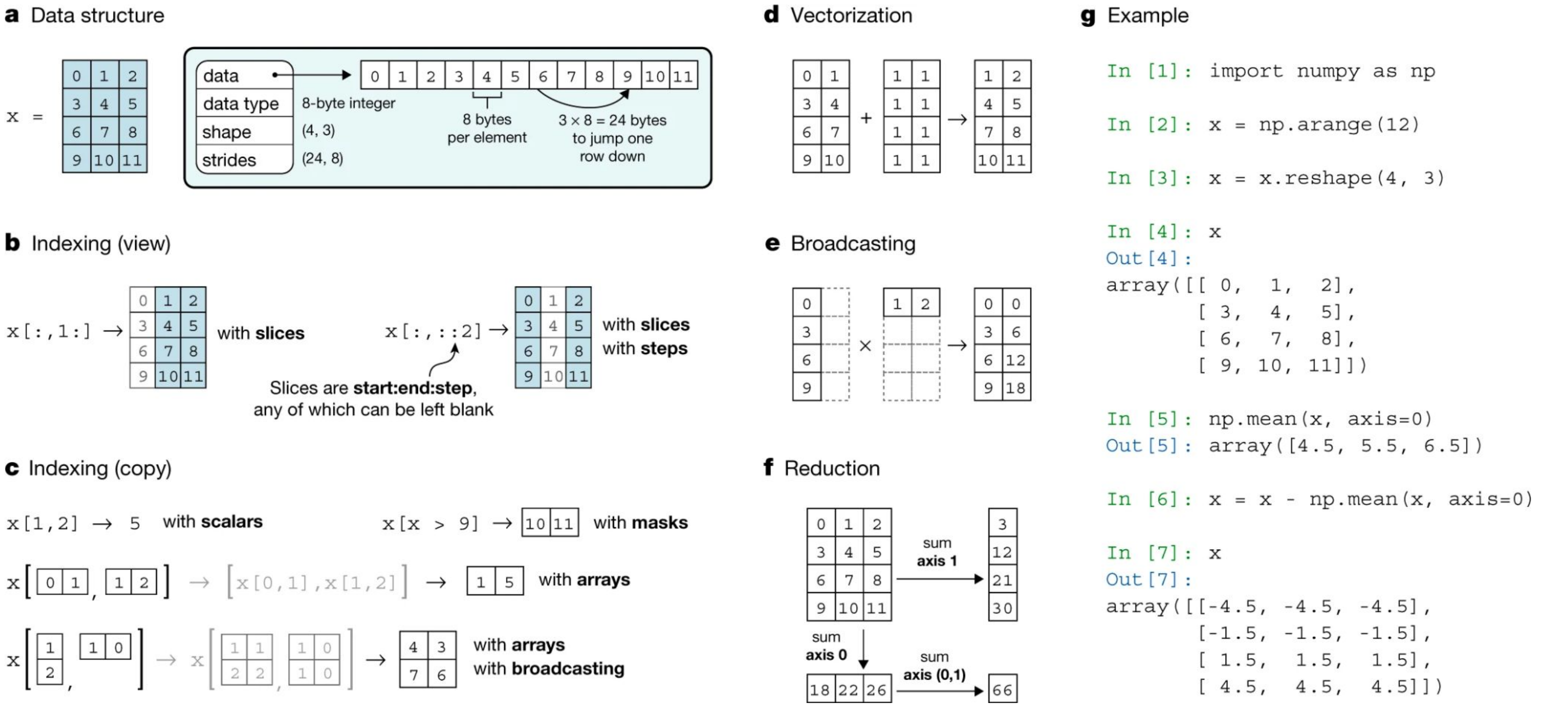
```
> python pi-dask.py
1.2273
```

Dask is very popular tool in Python community.

Good documentation with many examples and tips for performance.

Can scale out to multiple nodes! (need to do a little extra work to setup and connect to workers)

Array programming with NumPy



Harris, C.R., Millman, K.J., van der Walt, S.J. *et al.* Array programming with NumPy. *Nature* 585, 357–362 (2020). https://doi.org/10.1038/s41586-020-2649-2

Array programming with NumPy

pi-numpy.py

```
import time
import numpy as np

def estimate_pi(n):
    xy = np.random.uniform(0, 1, (n, 2))
    c = np.sum(np.linalg.norm(x, axis=1) < 1)
    return c * 4.0 / n

n = 20_000_000

start = time.time()
result = estimate_pi(n)
end = time.time()

print(end - start)
```

```
> python pi-numpy.py
0.4738
```

NumPy is the foundation of many scientific data processing libraries

Use array programming to get C-like performance in Python!

Indirect Parallelism in Python

```
import numpy as np

# construct a random symmetric positive definite matrix
n = 1000
b = np.random.rand(n, n)
a = b.T @ b

# compute eigenvalue decomposition
w, v = np.linalg.eigh(a)
```

Many linear algebra methods in NumPy use a BLAS backend such as OpenBLAS or Intel's MKL, which may use multiple threads.

The multithreading parallelism in lower level backends used by NumPy is not constrained by Python's GIL.

The OMP_NUM_THREADS environment variable can be used to control number of threads used by BLAS backends of NumPy

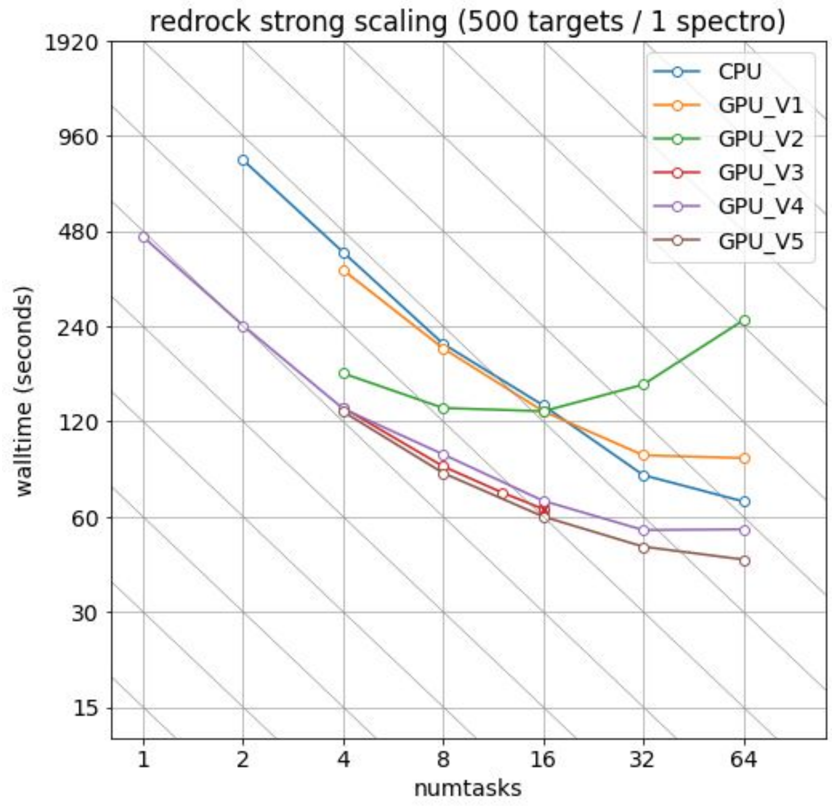
```
> python -m timeit -s "...[snip]..." "np.linalg.eigh(a)"
1 loop, best of 5: 427 msec per loop
```

Indirect Parallelism in Python

```
OMP_NUM_THREADS=1   | n=1000 | 2 loops, best of 5: 188 msec per loop
OMP_NUM_THREADS=2   | n=1000 | 2 loops, best of 5: 126 msec per loop
OMP_NUM_THREADS=4   | n=1000 | 2 loops, best of 5: 117 msec per loop
OMP_NUM_THREADS=8   | n=1000 | 2 loops, best of 5: 94.9 msec per loop
OMP_NUM_THREADS=16  | n=1000 | 2 loops, best of 5: 85.4 msec per loop
OMP_NUM_THREADS=32  | n=1000 | 5 loops, best of 5: 98.3 msec per loop
OMP_NUM_THREADS=64  | n=1000 |  1 loop, best of 5: 327 msec per loop
OMP_NUM_THREADS=128 | n=1000 |  1 loop, best of 5: 468 msec per loop
OMP_NUM_THREADS=256 | n=1000 |  1 loop, best of 5: 446 msec per loop
```

By default, the OpenMP runtime used by BLAS backends will typically use 1 thread per core.
There are 128 on cores on a Perlmutter CPU node.

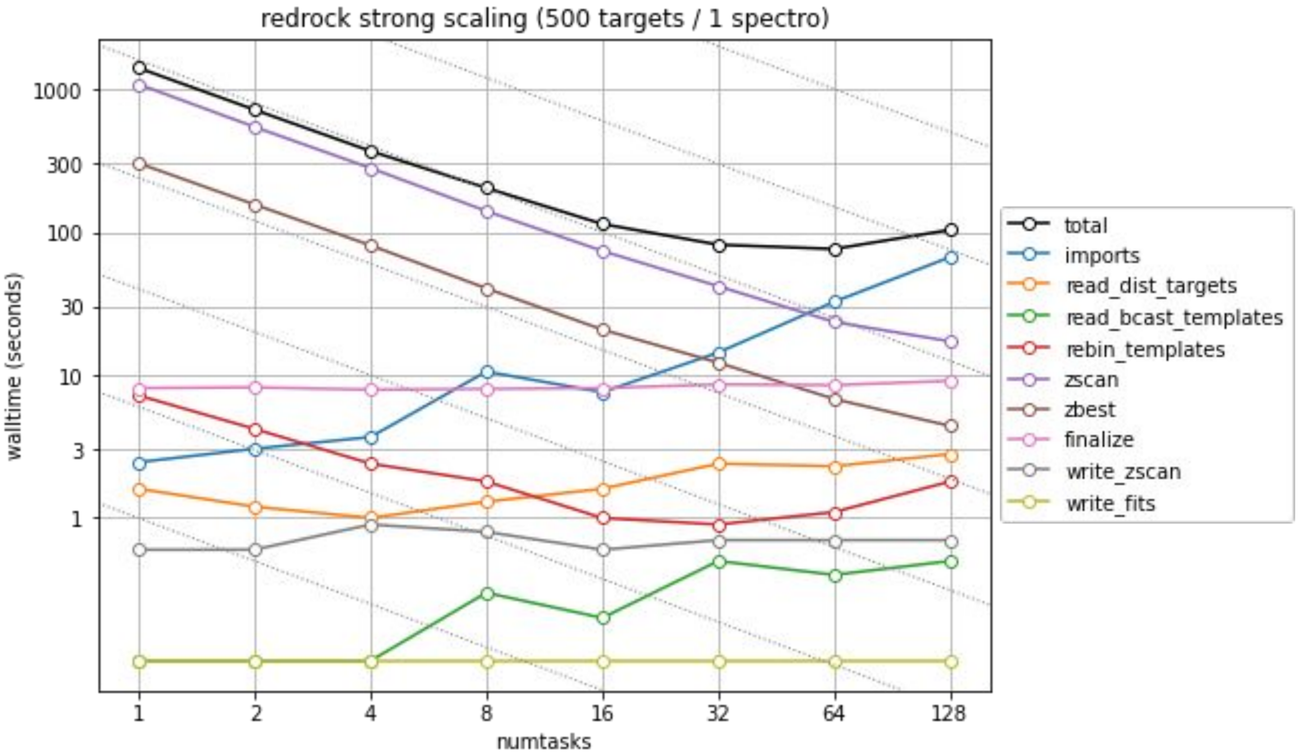
Scaling performance analysis

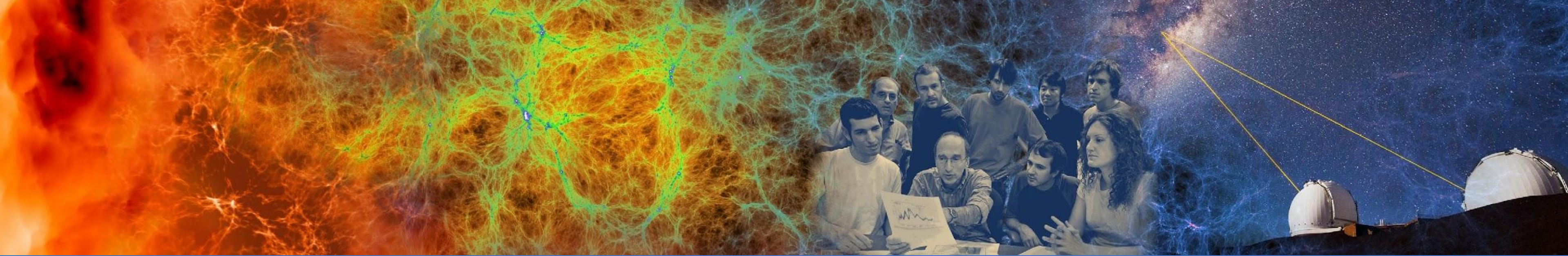


Name	Description	Notes
CPU	baseline	
GPU_V1	zchi2_batch on gpu	
GPU_V2	batch dot product of target.spectra.R with template bases	batch sizes become too small with many ranks
GPU_V3	remove distributed template redshift ranges	OOM above 16 ranks
GPU_V4	4 ranks use GPUs, all others are CPU only. lopsided distribution of work.	
GPU_V5	use mpi ranks to rebin templates and combine (partial undo of GPU_V3)	

NERSC BERKELEY LAB U.S. DEPARTMENT OF ENERGY Office of Science

Scaling performance analysis





Python + GPUs

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, PyOpenCL, PyCUDA, CUDA Python**
  - 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)
```

GPU programming in Python

```python
import cupy
import numba.cuda
import numpy

# CUDA kernel
@numba.cuda.jit
def _cuda_addone(x):
    i = numba.cuda.grid(1)
    if i < x.size:
        x[i] += 1

# convenience wrapper with thread/block
configuration
def addone(x):
    # threads per block
    tpb = 256
    # blocks per grid
    bpg = (x.size + (tpb - 1)) // tpb
    _cuda_addone[bpg, tpb](x)
```

https://docs.cupy.dev/en/stable/user_guide/basic.html
https://numba.readthedocs.io/en/stable/cuda/index.html

```python
# create array on device using cupy
x = cupy.zeros(10000)

# pass cupy ndarray to numba.cuda kernel
addone(x)

# Use numpy api with cupy ndarray
# (result is still on device)
total = numpy.sum(x)
```

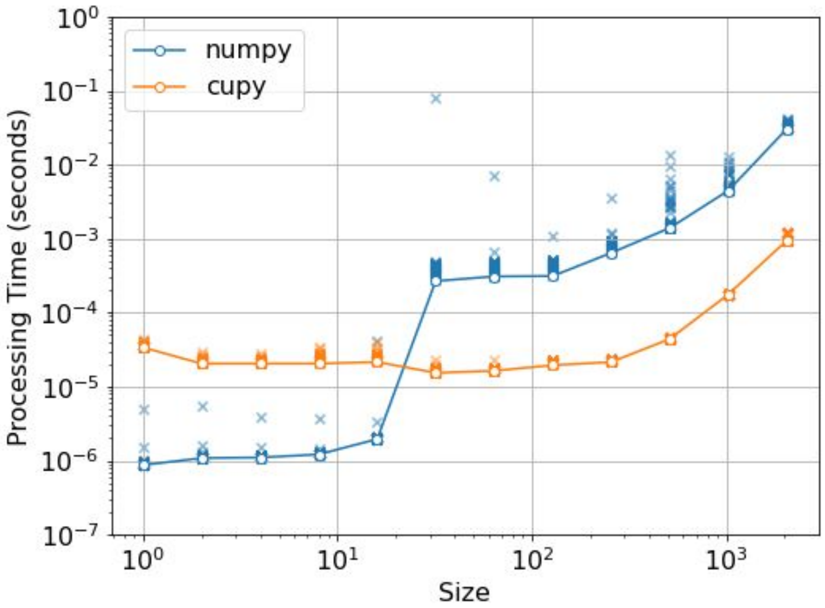
- NumPy's **__array_function__** protocol (NEP 18)
  - `numpy.sum(x) -> cupy.sum(x)`
- CPU and GPU execution paths can share same implementation (sometimes)
- Can also use helper functions to get the appropriate array module. For example:
  - `xp = cupy.get_array_module(x)`

Is my code a good fit for a GPU?

There's a good chance it is for cases where:

- operations can be performed on "large" arrays, matrices, images, etc
- IO is not a bottleneck

```
a = xp.random.rand(size, size)
b = xp.random.rand(size, size)
def f(a, b):
    return xp.dot(a, b)
```



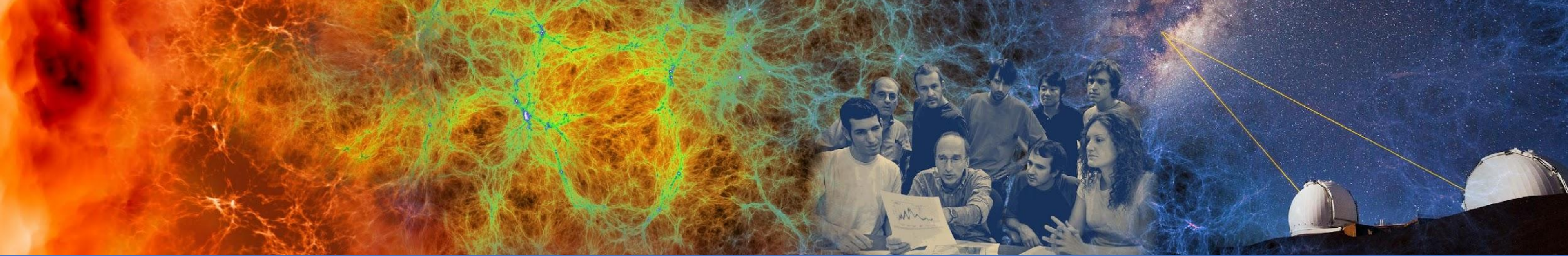
It can be "expensive" to move excessive amounts of data between device and host memory.

There is overhead for launching kernels on the GPU.

CPUs → low latency
GPUs → high throughput

Final thoughts

- Array Programming with NumPy!
  - eliminate for-loops in your program
  - vectorization / broadcasting / indexing
- Python startup is filesystem intensive. Containers may help with this at larger scales.
- You'll likely use more than one level of parallelism, consider composability of your choices.
- Profile your application before optimizing!
  - print/logging time differences is a good place to start



Thank you

Multithreading in Python