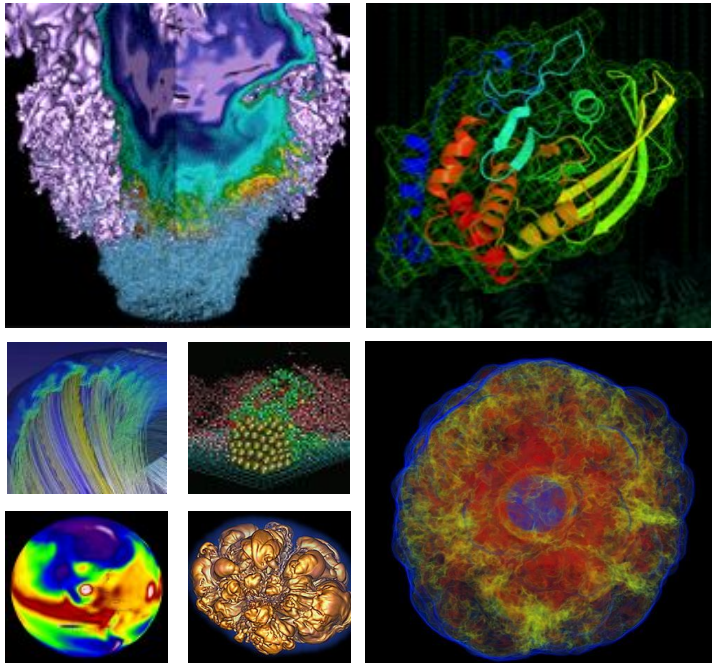
Optimizing Python-Based Spectroscopic Data Processing on NERSC Supercomputers



Laurie Stephey

SciPy 2019, July 10 2019

Rollin Thomas, NERSC and Stephen Bailey, DESI

Lawrence Berkeley National Laboratory

NERSC

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB

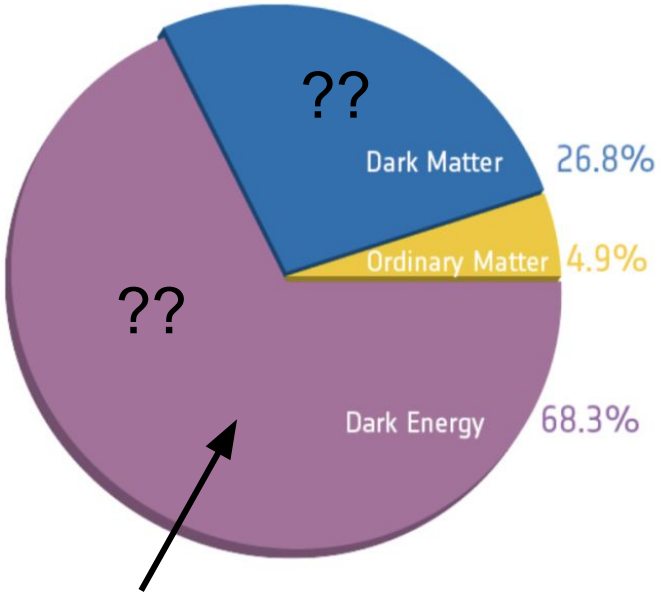
Overview of this talk

- DESI Python code now 5-7x faster!
- How?
  - Profiled to find hotspots
  - Major restructures
  - JIT compiled with Numba
- MPI vs Dask for parallelization (chose MPI)
- Current and future work: GPUs

What is DESI?

Dark Energy Spectroscopic Instrument

The Mayall telescope on Kitt Peak, AZ, where the DESI instrument is currently being installed



We currently believe the universe is almost 70 percent dark energy, but we don't have a good understanding of what dark energy actually is!

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB

A 3D map of the universe!

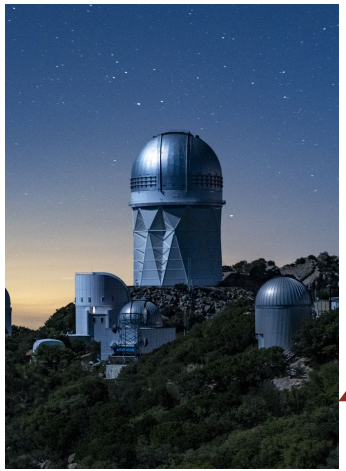
- Over 5 years, DESI will make the most detailed 3D map of the universe ever!
- They will use this map to better understand what dark energy is and its role throughout the evolution of the universe
- I am not a cosmologist, so for more (and better!) information, check out https://www.desi.lbl.gov/

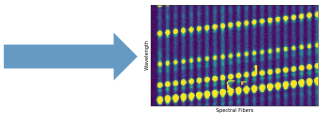
What is NERSC?

- Workhorse computing facility for the Office of Science in the US Department of Energy
- Located at Lawrence Berkeley Lab in Berkeley, California
- 7000 users
- Cori: current system (Intel Haswell + KNL)
- Perlmutter: future system (AMD CPUs + NVIDIA GPUs)

Why worry about speed?

DESI will take hundreds of images every night for 5 years





Now

2020

Will send images to NERSC to be quickly processed

Processed data used to make decisions for next night of observing

This will require many millions of CPU hours! Can we make it more efficient?

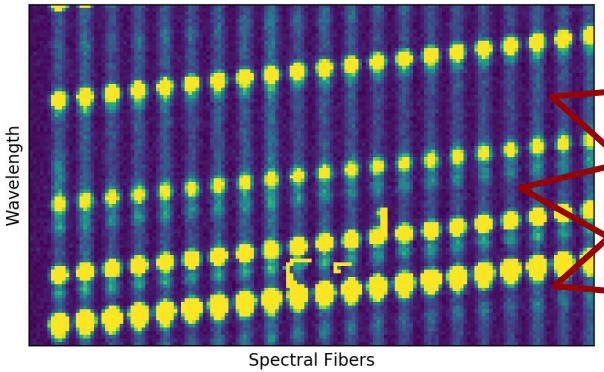
Project goals and constraints

- DESI commissioning starts this fall
- Need fast turnaround for survey operations and scientific results
- DESI prioritizes quick, clear, maintainable code → Python! (This audience gets it! 😀)
- Asked that we don't rewrite code in C or Cython (harder to write, read, and maintain)
- Our mission: make spectral extraction code faster!

What is spectral extraction?

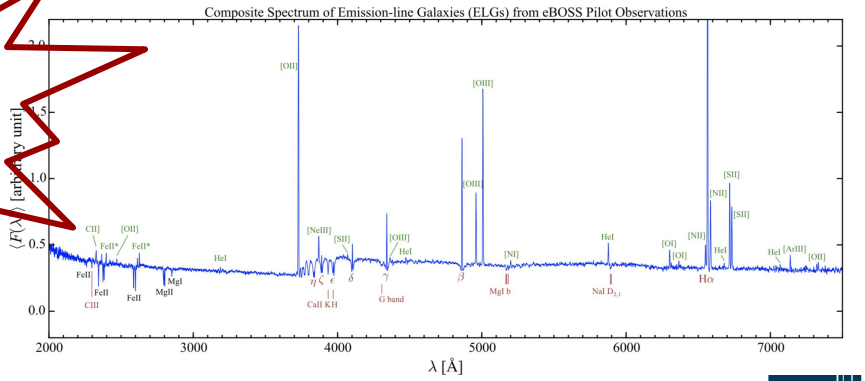
- Take raw image data and turn it into usable spectra
- “Spectroperfectionism” 2D fitting (expensive!)
- Eigenvalue decomposition
- Evaluating special functions
- Bookkeeping...

Raw data on CCD



Spectral extraction

Goal: High quality output spectra



Where do we begin?

desihub / specter

Unwatch 47 | Star 4 | Fork 3

Code | Issues 15 | Pull requests 0 | Projects 0 | Wiki | Insights

A toolkit for simulating multi-object spectrographs

551 commits | 6 branches | 25 releases | 5 contributors | View license

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

sbailey updated changes.rst for PR #71 faster pgh — Latest commit a70f99e 10 days ago

bin	add unicode_literals	2 years ago
dev	add bright star on border case to divide-and-conquer notebook	a year ago
doc	updated changes.rst for PR #71 faster pgh	10 days ago
etc	update module file	2 years ago
py/specter	Merge pull request #71 from desihub/fix_pgh	10 days ago
.coveragerc	adding top-level files	3 years ago
.gitignore	Specter speedup (#56)	10 months ago
.travis.yml	.travis.yml debugging	5 months ago
LICENSE.rst	fix license file	a year ago
MANIFEST.in	update MANIFEST.in	2 years ago
README.rst	remove incorrect versioning instructions	2 years ago
setup.py	update release notes and version for specter/0.8.2	10 months ago

Can you find the slow part?

Profiling tools: cProfile

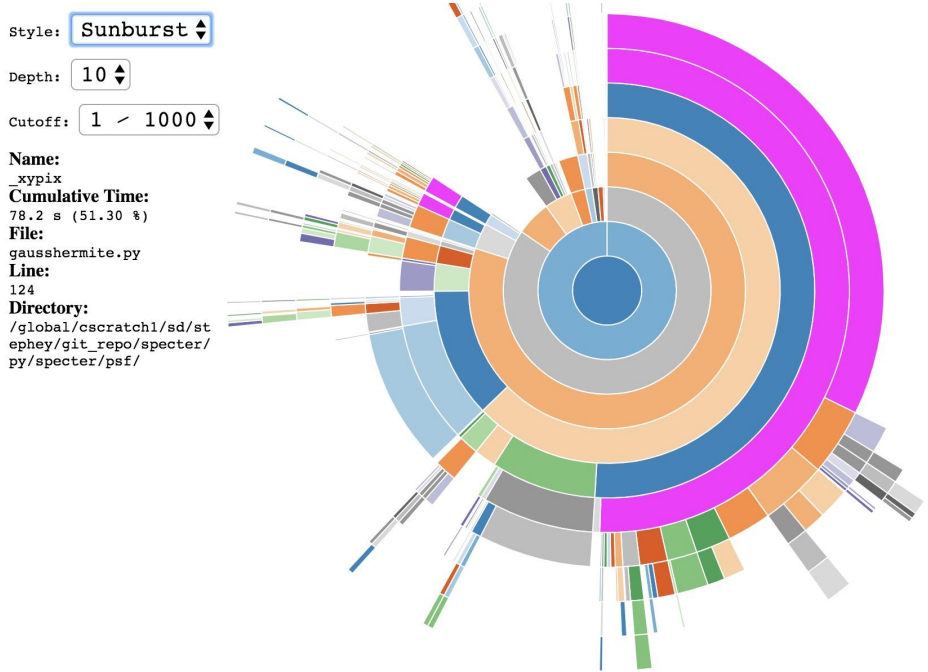
- **Easy! Built-in, no modifications to your code:**
- **Example:**

```
python -m cProfile -o output.prof path/to/your/script arg1 arg2
```

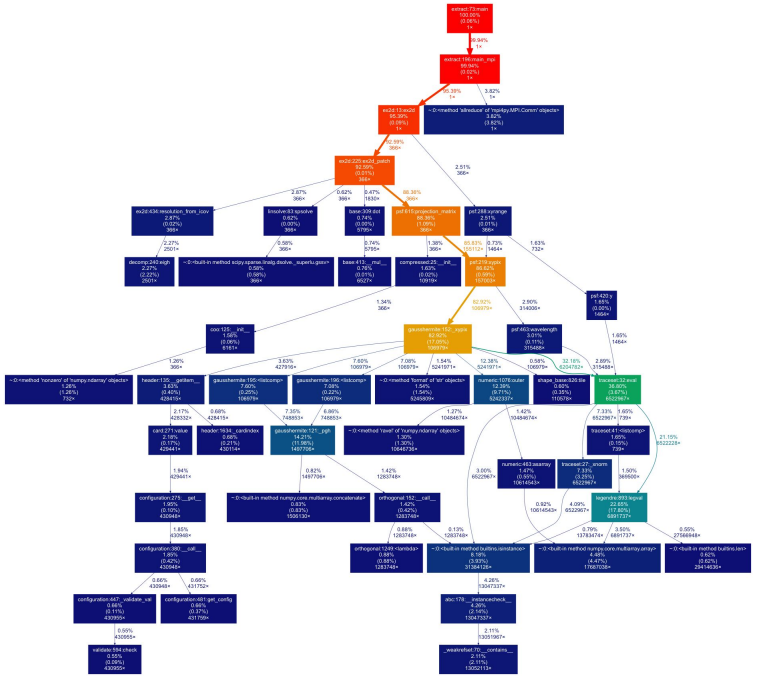
- **Writes human readable files, but nicer to visualize**

Profiling tools: cProfile

cProfile results visualized with Snakeviz and gprof2dot



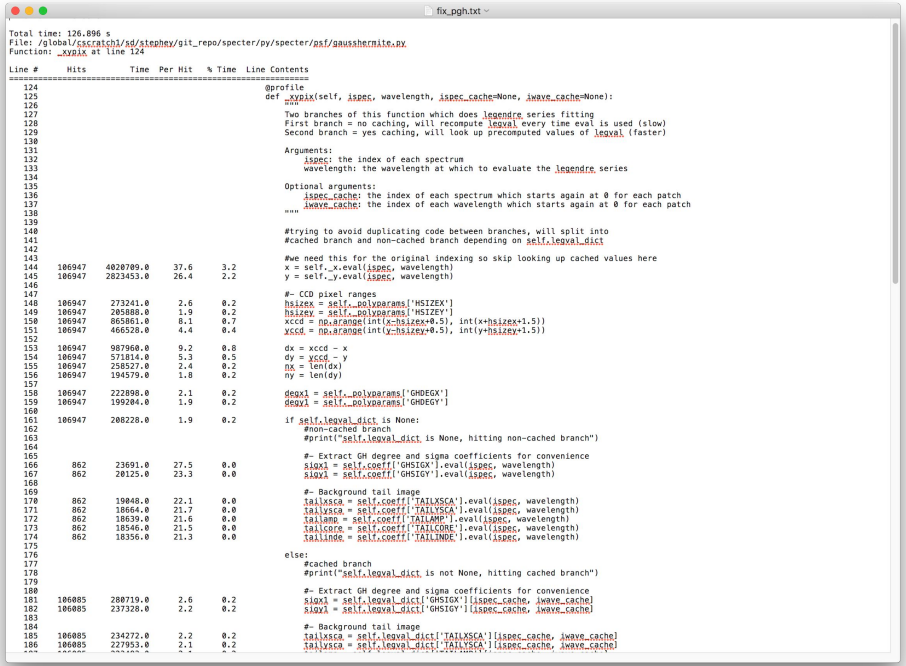
Snakeviz



gprof2dot

Profiling tools: line_profiler

- Manually decorate your functions of interest
- Get line-by-line profiling information



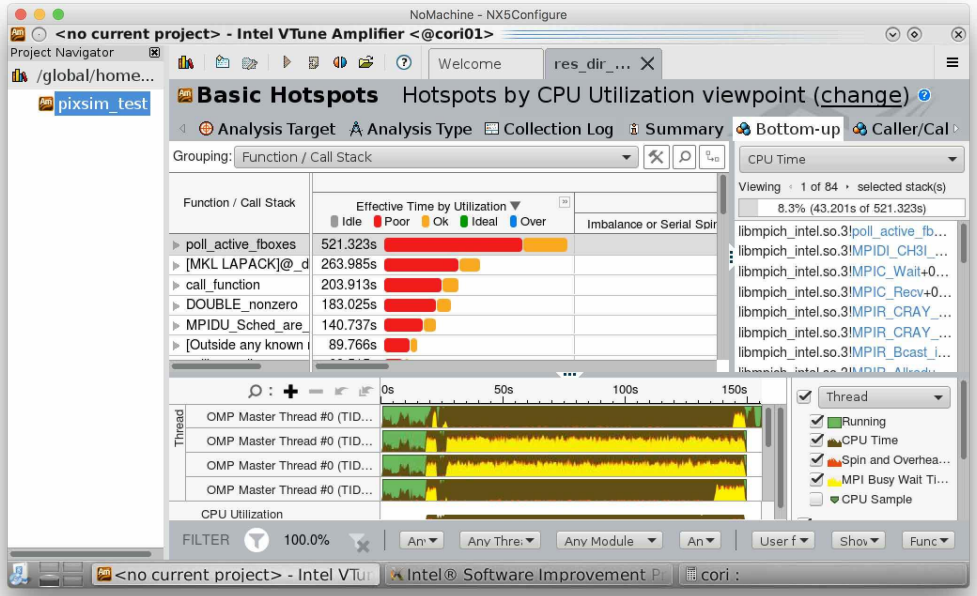
fix_pgh.txt

```
Total time: 126.896 s
File: /global/cscratch1/sd/stephey/git_repo/specter/py/specter/psf/gausshermite.py
Function: _xypix at line 124

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   124                                           @profile
   125                                           def _xypix(self, ispec, wavelength, ispec_cache=None, iwave_cache=None):
   126                                               """
   127                                               Two branches of this function which does legendre series fitting
   128                                               First branch = no caching, will recompute legval every time eval is used (slow)
   129                                               Second branch = yes caching, will look up precomputed values of legval (faster)
   130
   131                                               Arguments:
   132                                                   ispec: the index of each spectrum
   133                                                   wavelength: the wavelength at which to evaluate the legendre series
   134
   135                                               Optional arguments:
   136                                                   ispec_cache: the index of each spectrum which starts again at 0 for each patch
   137                                                   iwave_cache: the index of each wavelength which starts again at 0 for each patch
   138                                               """
   139
   140                                               #trying to avoid duplicating code between branches, will split into
   141                                               #cached branch and non-cached branch depending on self.legval_dict
   142
   143                                               #we need this for the original indexing so skip looking up cached values here
   144    106947    4020709.0     37.6      3.2       x = self._x.eval(ispec, wavelength)
   145    106947    2823453.0     26.4      2.2       y = self._y.eval(ispec, wavelength)
   146
   147                                               #- CCD pixel ranges
   148    106947     273241.0      2.6      0.2       hsizex = self._polyparams['HSIZEX']
   149    106947     205888.0      1.9      0.2       hsizey = self._polyparams['HSIZEY']
   150    106947     865861.0      8.1      0.7       xccd = np.arange(int(x-hsizex+0.5), int(x+hsizex+1.5))
   151    106947     466528.0      4.4      0.4       yccd = np.arange(int(y-hsizey+0.5), int(y+hsizey+1.5))
   152
   153    106947     987960.0      9.2      0.8       dx = xccd - x
   154    106947     571814.0      5.3      0.5       dy = yccd - y
   155    106947     258527.0      2.4      0.2       nx = len(dx)
   156    106947     194579.0      1.8      0.2       ny = len(dy)
   157
   158    106947     222898.0      2.1      0.2       degx1 = self._polyparams['GHDEGX']
   159    106947     199204.0      1.9      0.2       degy1 = self._polyparams['GHDEGY']
   160
   161    106947     208228.0      1.9      0.2       if self.legval_dict is None:
   162                                                   #non-cached branch
   163                                                   #print("self.legval_dict is None, hitting non-cached branch")
   164
   165                                                   #- Extract GH degree and sigma coefficients for convenience
   166       862      23691.0     27.5      0.0           sigx1 = self.coeff['GHSIGX'].eval(ispec, wavelength)
   167       862      20125.0     23.3      0.0           sigy1 = self.coeff['GHSIGY'].eval(ispec, wavelength)
   168
   169                                                   #- Background tail image
   170       862      19048.0     22.1      0.0           tailxsca = self.coeff['TAILXSCA'].eval(ispec, wavelength)
   171       862      18664.0     21.7      0.0           tailysca = self.coeff['TAILYSCA'].eval(ispec, wavelength)
   172       862      18639.0     21.6      0.0           tailamp = self.coeff['TAILAMP'].eval(ispec, wavelength)
   173       862      18546.0     21.5      0.0           tailcore = self.coeff['TAILCORE'].eval(ispec, wavelength)
   174       862      18356.0     21.3      0.0           tailinde = self.coeff['TAILINDE'].eval(ispec, wavelength)
   175
   176                                               else:
   177                                                   #cached branch
   178                                                   #print("self.legval_dict is not None, hitting cached branch")
   179
   180                                                   #- Extract GH degree and sigma coefficients for convenience
   181    106085     280719.0      2.6      0.2           sigx1 = self.legval_dict['GHSIGX'][ispec_cache, iwave_cache]
   182    106085     237328.0      2.2      0.2           sigy1 = self.legval_dict['GHSIGY'][ispec_cache, iwave_cache]
   183
   184                                                   #- Background tail image
   185    106085     234272.0      2.2      0.2           tailxsca = self.legval_dict['TAILXSCA'][ispec_cache, iwave_cache]
   186    106085     227953.0      2.1      0.2           tailysca = self.legval_dict['TAILYSCA'][ispec_cache, iwave_cache]
```

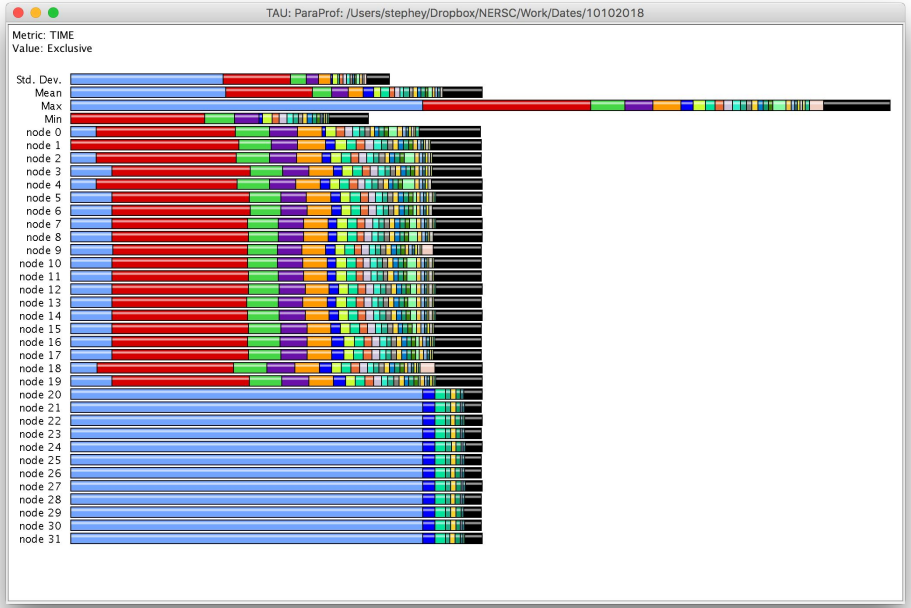
Profiling tools: Intel Vtune

- Vtune has some support for Python
- Tricky to use, collect, and display the right data



Profiling tools: Tau (U Oregon)

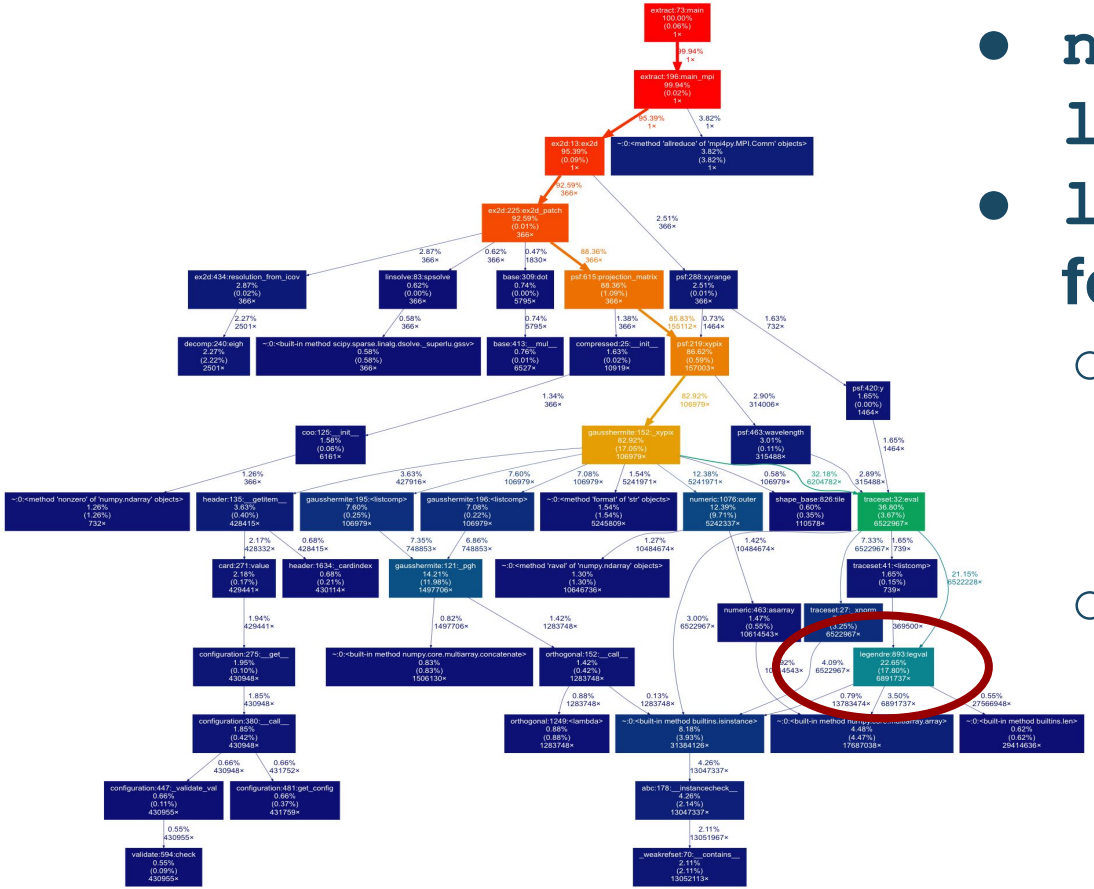
- For best results, custom build for your application
- Excellent and interactive visualizations
- Great for MPI applications



Our final profiling strategy

- Start by profiling the code with cProfile, visualize
- Identify a hotspot
- Examine the function with line_profiler to get more information
- Decide on a strategy for the hotspot:
  - Speed up function by JIT-compiling (easier)
  - Restructure the code (harder)
- Benchmark and reprofile the code, see how well it worked (or didn't)
- Rinse and repeat, use "big" tools only when required

Hotspot: `legval`



- `numpy.polynomial.legendre.legval`
- `legval` was a problem for two reasons:
  - Expensive function (in Python rather than C)
  - Called too many times (often with scalar values!)

Strategy #1 JIT compile

- Numba is a JIT compiler for Python (LLVM under the hood)
- Get the function into a form Numba will accept (can be tricky)
  - Remove type checking, other unsupported functionality
  - Remove all SciPy (not currently supported) 😞
  - Size and type of arrays must be known at compile time
- Function is now faster!

JIT compile `legval`

Before

```
def legval(x, c, tensor=True):
    c = np.array(c, ndmin=1, copy=0)
    if c.dtype.char in '?bBhHiIlLqQpP':
        c = c.astype(np.double)
    if isinstance(x, (tuple, list)):
        x = np.asarray(x)
    if isinstance(x, np.ndarray) and tensor:
        c = c.reshape(c.shape + (1,)*x.ndim)

    if len(c) == 1:
        c0 = c[0]
        c1 = 0
    elif len(c) == 2:
        c0 = c[0]
        c1 = c[1]
    else:
        nd = len(c)
        c0 = c[-2]
        c1 = c[-1]
        for i in range(3, len(c) + 1):
            tmp = c0
            nd = nd - 1
            c0 = c[-i] - (c1*(nd - 1))/nd
            c1 = tmp + (c1*x*(2*nd - 1))/nd
    return c0 + c1*x
```

Removed this, Numba won't allow type checking

Removed these cases since we know len(c) > 2

After

Add Numba decorator

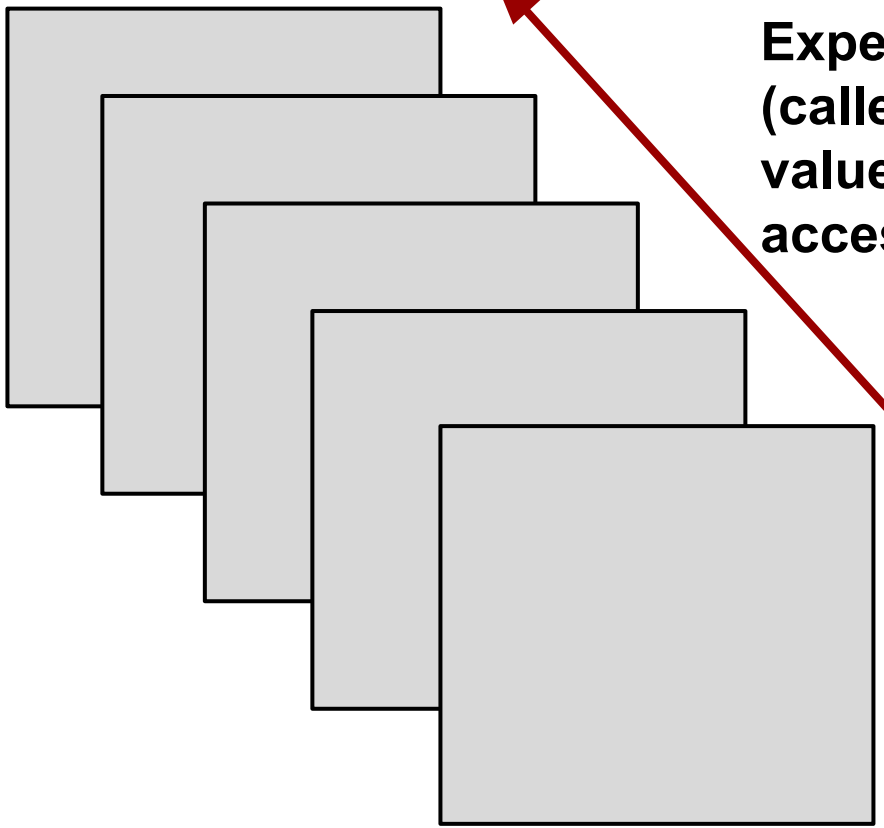
```
@numba.jit(nopython=True,cache=False)
def legval_numba(x, c):
    nd=len(c)
    ndd=nd
    xlen = x.size
    c0=c[-2]*np.ones(xlen)
    c1=c[-1]*np.ones(xlen)
    for i in range(3, ndd + 1):
        tmp = c0
        nd = nd - 1
        nd_inv = 1/nd
        c0 = c[-i] - (c1*(nd - 1))*nd_inv
        c1 = tmp + (c1*x*(2*nd - 1))*nd_inv
    return c0 + c1*x
```

Added extra info so the arrays don't change size

Some changes → 16x speedup on KNL, 5x speedup on Haswell for this function!

Strategy #2: Restructure

High level



Expensive function (called fewer times, values stored and accessed later)

Low level

Expensive function (called many times)

Restructure to mitigate `legval`

Before

```
#- Extract GH degree and sigma coefficients for convenience
sigx1 = self.coeff['GHSIGX'].eval(ispec, wavelength)
sigy1 = self.coeff['GHSIGY'].eval(ispec, wavelength)

#- Background tail image
tailxsca = self.coeff['TAILXSCA'].eval(ispec, wavelength)
tailysca = self.coeff['TAILYSCA'].eval(ispec, wavelength)
tailamp = self.coeff['TAILAMP'].eval(ispec, wavelength)
tailcore = self.coeff['TAILCORE'].eval(ispec, wavelength)
tailinde = self.coeff['TAILINDE'].eval(ispec, wavelength)
```

Calling `eval` (which calls `legval`) again and again for scalars

After

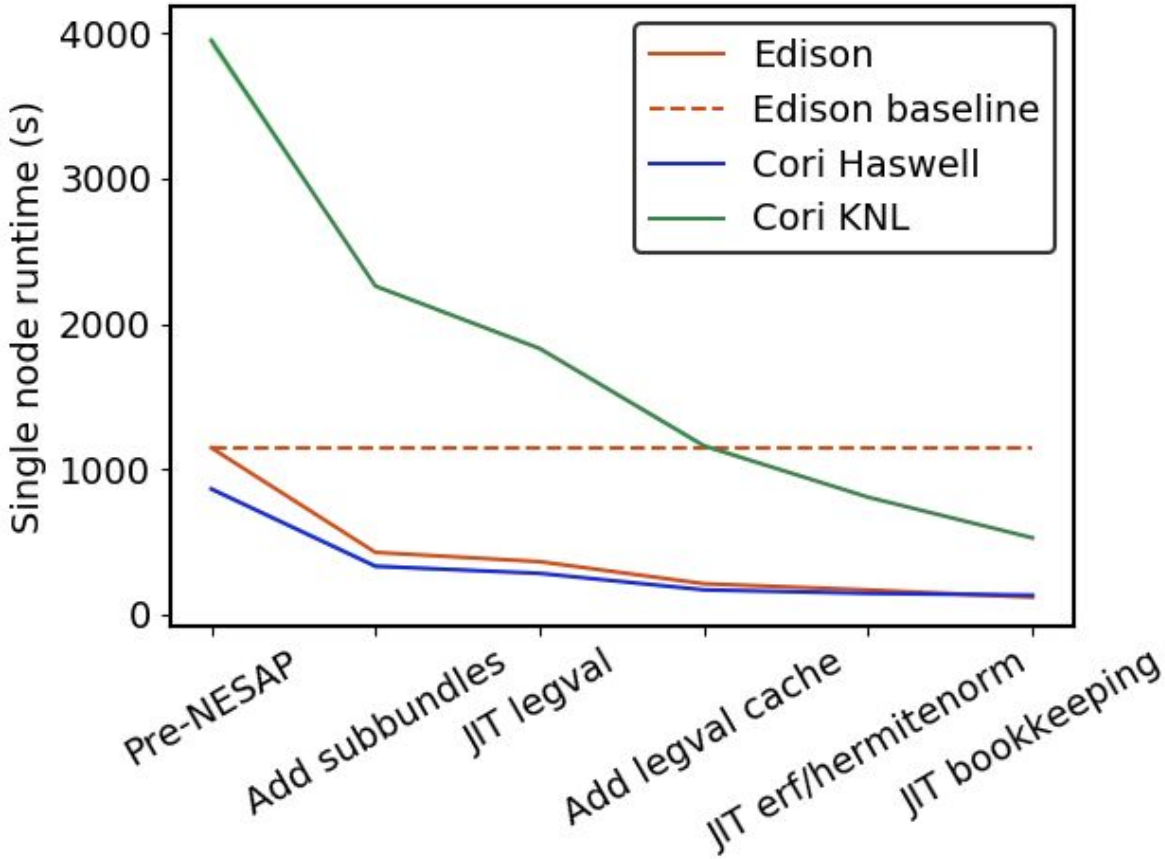
```
#- Extract GH degree and sigma coefficients for convenience
sigx1 = self.legval_dict['GHSIGX'][ispec_cache, iwave_cache]
sigy1 = self.legval_dict['GHSIGY'][ispec_cache, iwave_cache]

#- Background tail image
tailxsca = self.legval_dict['TAILXSCA'][ispec_cache, iwave_cache]
tailysca = self.legval_dict['TAILYSCA'][ispec_cache, iwave_cache]
tailamp = self.legval_dict['TAILAMP'][ispec_cache, iwave_cache]
tailcore = self.legval_dict['TAILCORE'][ispec_cache, iwave_cache]
tailinde = self.legval_dict['TAILINDE'][ispec_cache, iwave_cache]
```

Just looking up `legval` values we have already computed!

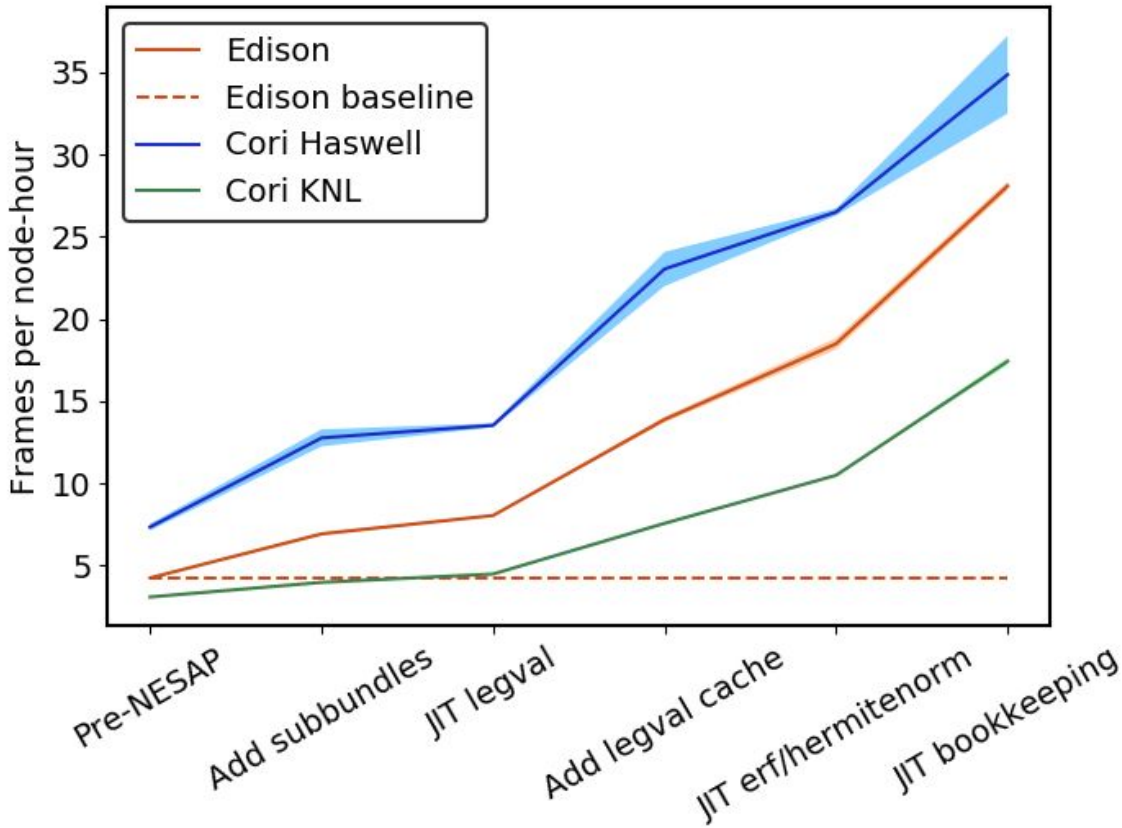
Is it faster? Yes!

DESI spectral extraction now 7-10x faster for a single node

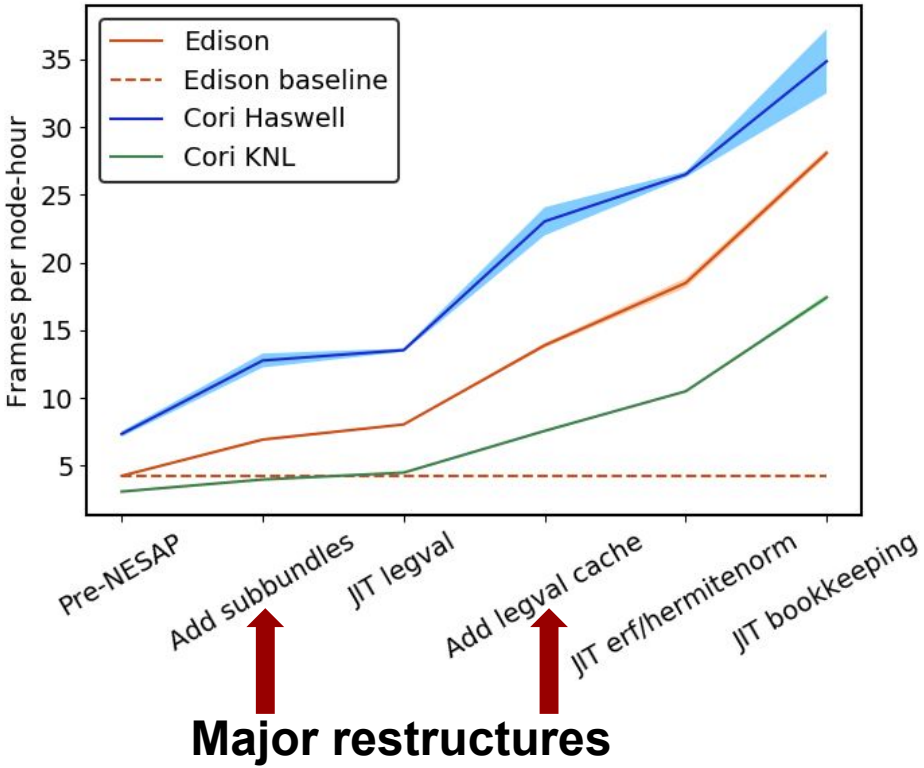


5-7x more throughput!

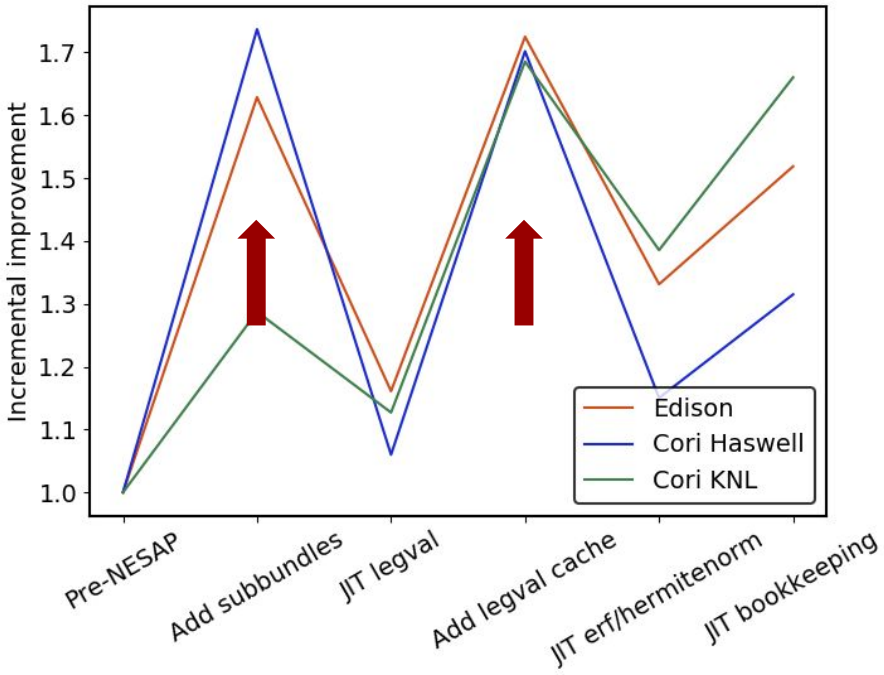
- For DESI, throughput is more meaningful than sheer speed
- How many frames can they process per CPU-hour?
- **5-7x more throughput**
- **Before: 33 million CPU hours**
- **After: 6.5 million CPU hours**



Restructures → more speedup

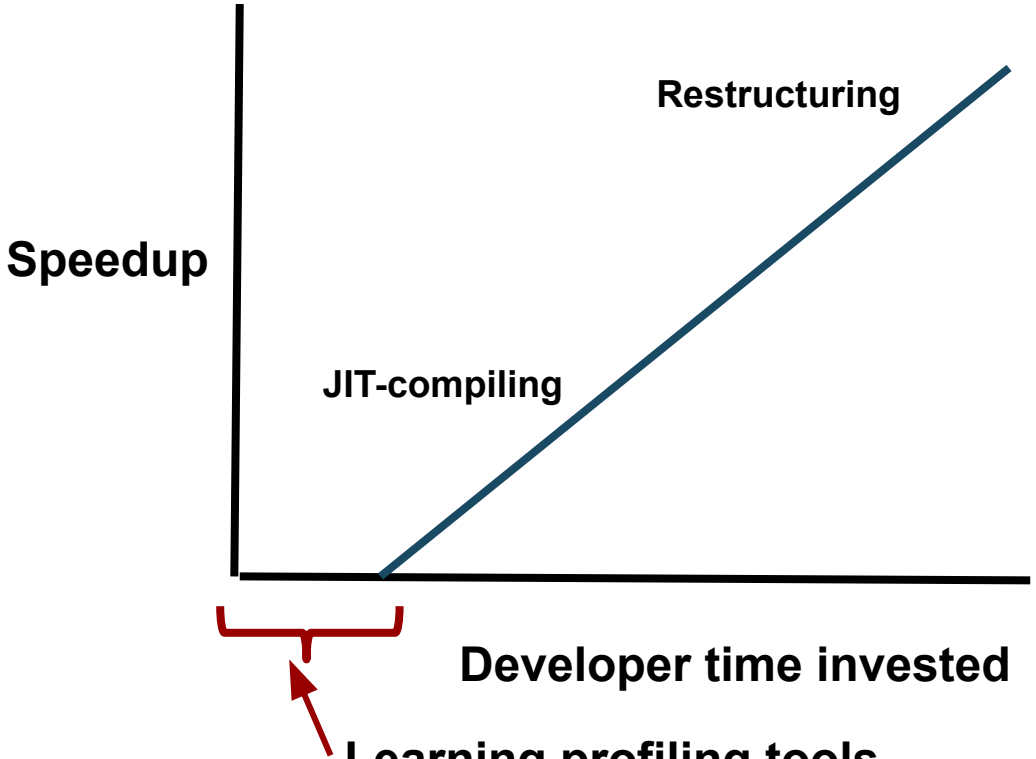


Major restructures



Restructures resulted in more incremental speedup than JIT-based optimizations

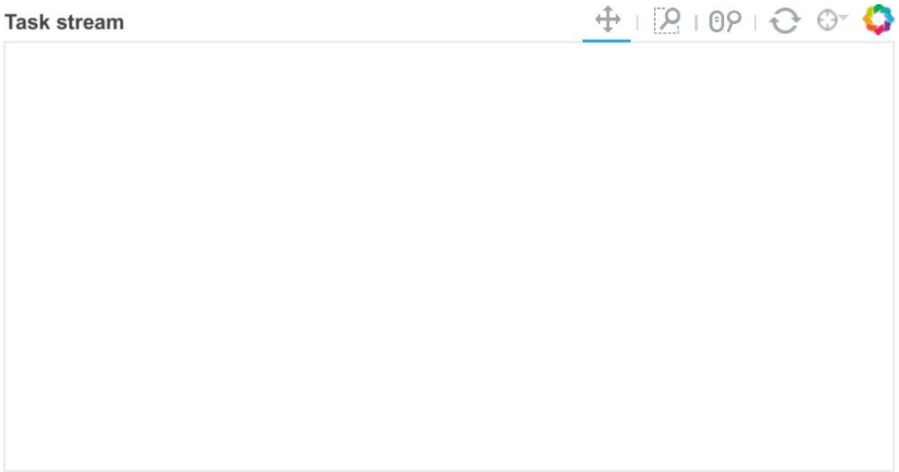
Our experience: time in ∝ speedup out



- Ultimate speedup is proportional to time invested (i.e. speedup is not a step function)
- ***No actual data here, just our experience!
- tl;dr: If you don't have much time, you can still obtain modest improvements

What about Dask?

- We like Dask and wondered if the DESI mpi4py could be replaced with Dask
- Short answer: no (too late!)

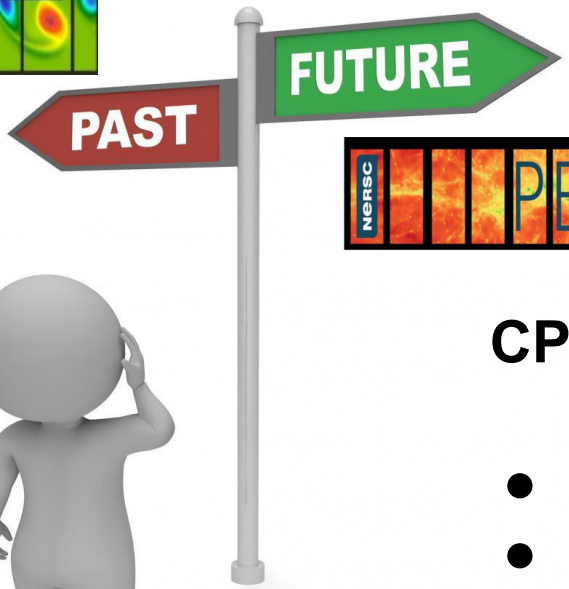


GIF reproduced from: https://distributed.dask.org/en/latest/web.html

“Change is the only constant in life”

- Heraclitus, Greek philosopher

CPUs



CPUs + GPUs

- **Coming in 2020**
- **Time to start preparing applications now**

U.S. DEPARTMENT OF ENERGY | Office of Science

Image reproduced from: https://freerangestock.com/photos/78641/past-future-sign-indicates-long-ago-and-destiny-3d-rendering.html

BERKELEY LAB

DESI GPU open questions

- How can DESI best leverage the GPUs?
- Should we focus on speeding up functions like `legval` in a bottom-up approach like we did for CPUs?
- Should we take a top-down approach and rethink our core algorithms? (Probably)
- What is the best option for Python? CuPy + Numba?
- This is our current/future work

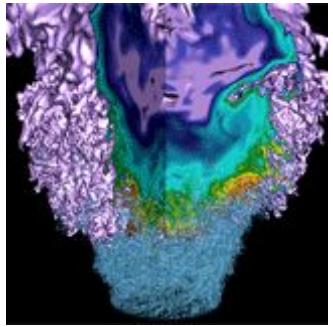
Summary

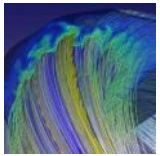
- We were able to help DESI improve their throughput 5-7x without rewriting in C!
- 33 million cpu hours → 6.5 million cpu hours
- Used two strategies to obtain speedup:
  - JIT-compiling with Numba (easy, but less overall improvement)
  - Restructuring of code (hard, but more overall improvement)
- Future work: port DESI code to GPUs

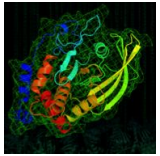
Want to know more?

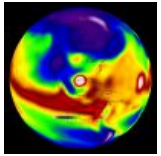
- Check out our documentation for profiling your Python code: https://docs.nersc.gov/programming/high-level-environments/python/profiling-python/
- Check out the DESI code on github: https://github.com/desihub
- Check out our SciPy 2019 proceedings paper: http://conference.scipy.org/proceedings/scipy2019/laurie_stephey.html

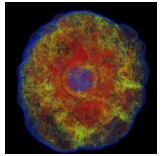
Thank you!

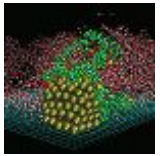










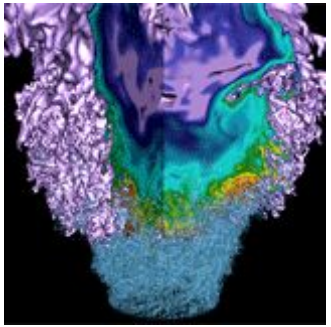


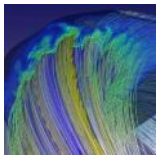
NERSC

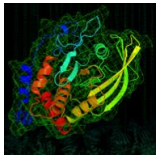
U.S. DEPARTMENT OF ENERGY | Office of Science

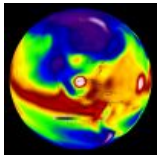
BERKELEY LAB

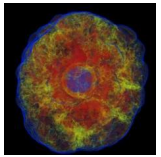
Extra slides

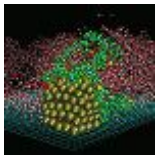






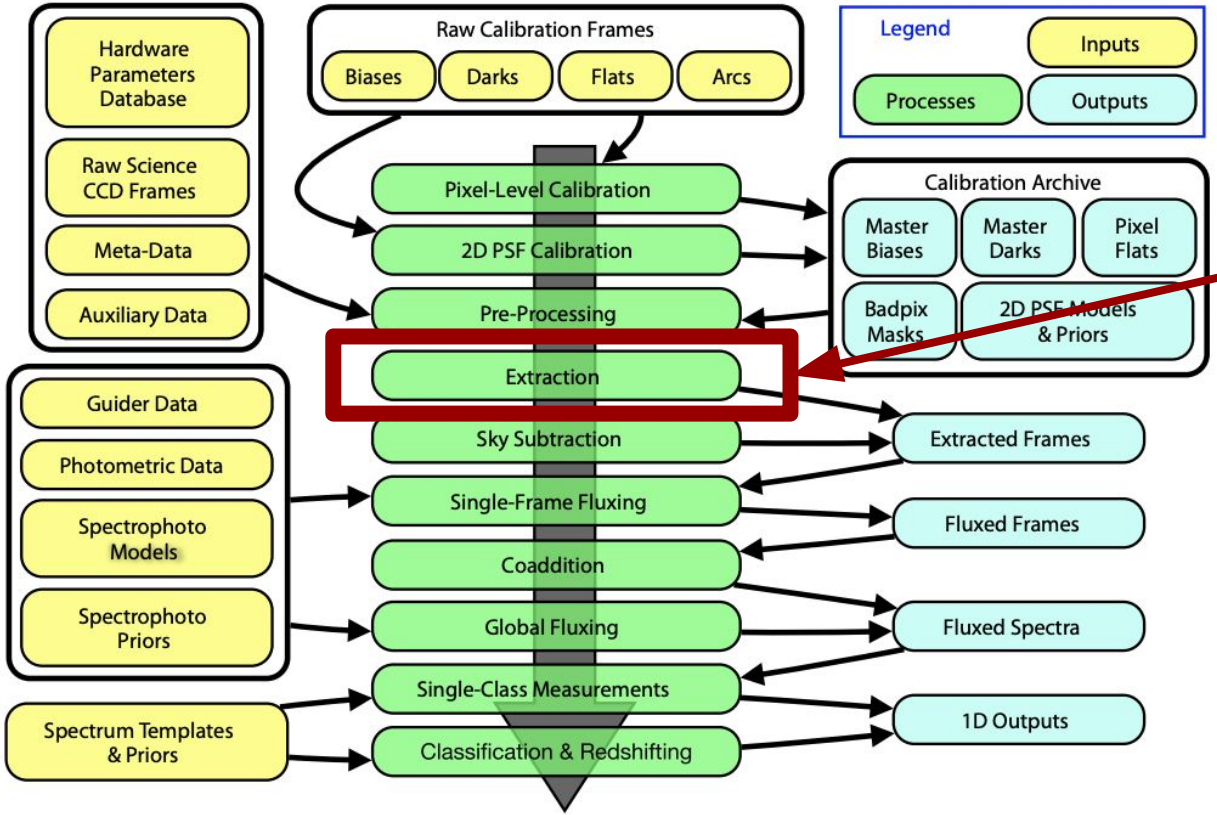






NERSC

The DESI pipeline

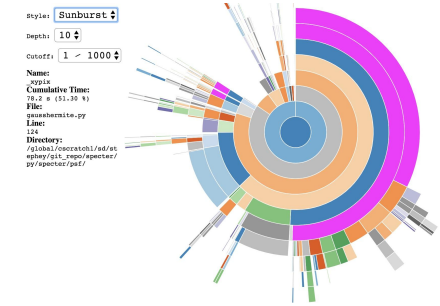


- The spectral extraction part of the pipeline was known to be a major bottleneck
- Our mission: speed it up!

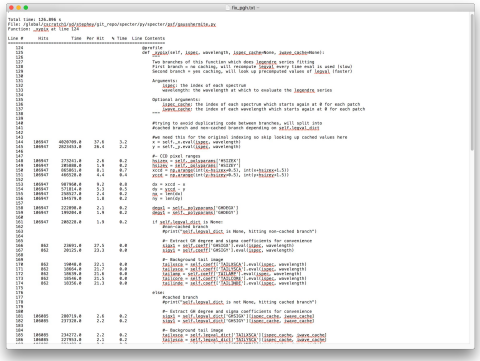
Profiling workflow: easy to hard

cProfile (built into Python)

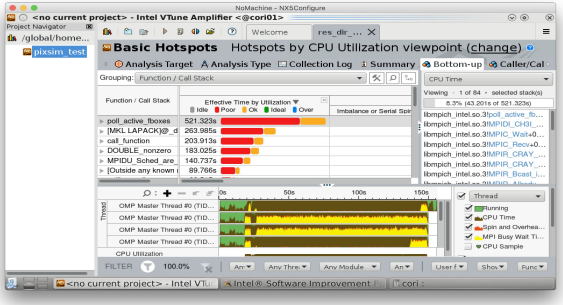
Snakeviz



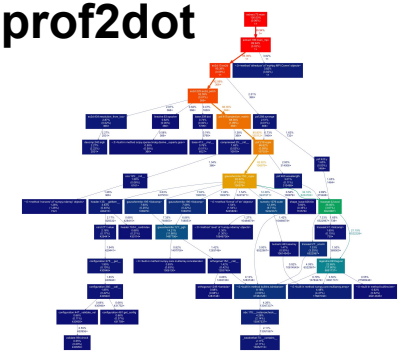
line_profiler



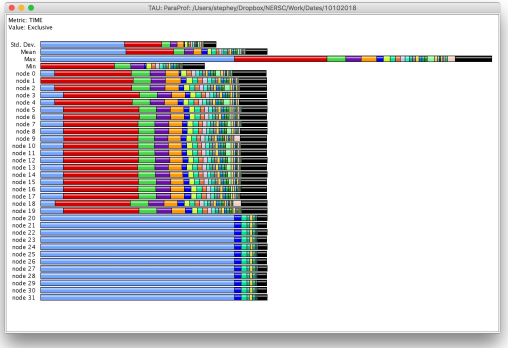
Intel Vtune



gprof2dot



Tau



Easy ⟷ **Hard**

DASK wasn't the right fit for DESI

- DESI uses mpi4py for MPI parallelism
- If DESI were using the concurrent futures API (MPI comm.spawn), it would have been conceptually similar to DASK and easier to transition
- Alas, this is not yet supported on NERSC Cori (due to Slurm compatibility)
- Ultimately we decided that changing to DASK was impractical for DESI
- Starting a new project with DASK in mind would be better than trying to adapt an established project

CuPy and Numba

- We have tested CuPy and Numba for GPU Python
- CuPy:
  - Almost a drop-in replacement for NumPy
  - Easy to use
  - Not all NumPy/SciPy is implemented
- Numba:
  - Harder to use than its CPU cousin
  - Most NumPy not allowed
  - Code ends up looking more like CUDA than Python

Testing `eigh` in CuPy

- We wanted to see how the `np.linalg.eigh` function scaled on an NVIDIA Volta GPU
- Eventually faster than the Cori CPUs, but only at large matrix size

CPU

```
import numpy as np

cpu_result =
np.linalg.eigh(data)
```

GPU

```
import cupy as cp

#move data to gpu
gpu_data = cp.asarray(data)

gpu_result =
cp.linalg.eigh(gpu_data)

#move data to cpu
cpu_result =
cp.asnumpy(gpu_result)
```